Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado em Engenharia Informática
2º Semestre, 2007/2008

De Arquitecturas Organizacionais em i* para Arquitecturas Baseadas em Agentes: Uma Abordagem Orientada a Modelos

Pedro Miguel Ribeiro Quendera Dias – Nº 26175

Orientador: Prof. Doutor João Araújo

Co-Orientadores: Prof. Dra. Ana Moreira, Prof. Dra. Carla Silva

31 de Julho de 2008

Nº do aluno: 26175
Nome: Pedro Miguel Ribeiro Quendera Dias

Título da dissertação:
De arquitecturas organizacionais em i* para arquitecturas baseadas em Agentes:
Uma abordagem orientada a modelos.

Palavras-Chave:

- Arquitectura Orientada a Modelos (AOM)
- Engenharia Orientada a Modelos (EOM)
- Transformações de Modelos
- i*
- Modelos Arquitecturais

Keywords:

- Model Driven Architecture (MDA)
- Model Driven Engineering (MDE)
- Model Transformations
- i*
- Architectural Models

Agradecimentos

Quero aproveitar este espaço para agradecer aos meus pais e ao meu tio, por nunca me terem deixado desistir, mesmo quando a minha cabeça estava em todo o lado menos na faculdade.

À minha namorada, Vera Isa, pela estabilidade emocional ao longo desta caminhada.

Um enorme obrigado ao Capitão, ao Conquistador e ao Campino por momentos inesquecíveis. Alguns até inexplicáveis. Juntos formámos (e ainda estamos aí para as curvas) uma equipa imparável.

Aos meus colegas de faculdade que me acompanharam e ajudaram ao longo destes longos anos de percurso académico. Em particular aos "Culelos", que me acolheram no seio do seu grupo, numa fase em que os da minha geração já tinham, por uma razão ou por outra, deixado a FCT.

Também quero deixar o meu obrigado ao Vasco Sousa e ao João Pedro Santos pela ajuda que me deram durante esta dissertação.

Aos meus orientadores um grande obrigado pela cooperação durante este ano importante da minha vida.

Resumo

Os requisitos dizem, normalmente, o que um sistema deve fazer, por oposição a como fazê-lo. O contexto organizacional justifica e ajuda a compreender os porquês que levam à necessidade de certos requisitos importantes para um sistema de software ser bem sucedido. As técnicas de modelação de requisitos oferecem o conhecimento que permite a análise necessária nesta fase inicial do desenvolvimento. Contudo, a maioria das técnicas de requisitos são mais adequadas para uma fase posterior do processo da engenharia de requisitos.

O desenvolvimento de software orientado a agentes procura preencher esta lacuna, mas é um paradigma relativamente recente. Para a consolidação deste novo paradigma, o projecto Tropos está a desenvolver uma abordagem para o desenvolvimento orientado a Agentes que se baseia nos conceitos sociais e intencionais oferecidos pela abordagem de modelação organizacional i. No entanto, o uso do i* não é suficientemente expressivo como uma linguagem de descrição arquitectural.*

Reconhecendo no UML a capacidade de actuar como linguagem de descrição arquitectural, esta dissertação especifica as transformações necessárias dos modelos arquitecturais organizacionais descritos em i, para os modelos arquitecturais descritos no perfil de Agência do UML utilizado para modelar sistemas multi-agente, através de uma abordagem orientada a modelos.*

Abstract

Requirements are usually understood as stating what a system is supposed to do, as opposed to how it should do it. The understanding of the organizational context justifies and helps understanding the rationales (the "Whys") that lead up to some important system requirements of a successful software system. Requirements modeling techniques can be used to help dealing with the knowledge and analysis needed in this early phase of the development. However, most existing requirements techniques are intended more for the later phase of requirements engineering.

Agent-oriented development aims at fulfilling this gap but this is quite recent. Among the several concerns required for the consolidation of this new paradigm, the Tropos project is developing an approach for agent-oriented development based on the social and intentional concepts offered by the i organizational modeling approach. However, the use of i* is not suitable as an architectural description language.*

Recognizing that the Unified Modeling Language (UML) can be extended to act as an architectural description language, this dissertation main goal is to design and automate the needed transformations between architectural models written in i to architectural models written in an Agency profile of UML, used to model multi-agent systems, using an MDE approach.*

Índice

Lista de Figuras

Lista de Tabelas

1 – Introdução

Este Capítulo aborda o problema que serviu de base à elaboração desta dissertação. Assim, introduzimos alguns conceitos chave para a nossa proposta, a motivação e aquela que vai ser a nossa contribuição para a comunidade científica na área da Engenharia do Software.

1.1 Engenharia de requisitos e Arquitectura de Software

A Engenharia de Requisitos é alvo de interesse por parte da comunidade científica na área de Informática. Isto acontece porque ela faz parte dos passos iniciais do desenvolvimento de software, e esses primeiros passos são importantes para o desenvolvimento, utilização e evolução de um programa ou sistema com qualidade. De acordo com Bubenko, à medida que aumenta a importância dos sistemas de computadores dentro das organizações, torna-se cada vez mais importante dar atenção aos passos iniciais da Engenharia de Requisitos [Bubenko, 1995].

Muita da investigação feita nesta área assenta nas primeiras fontes de requisitos, que expressam os desejos dos clientes acerca do que o sistema deve ser capaz de fazer. Os requisitos iniciais expressam os desejos dos clientes e são muitas vezes ambíguos, incompletos, inconsistentes e habitualmente expressos informalmente. A necessidade, por parte dos sistemas de grande porte, de fiabilidade e de reutilização, faz com que o uso de notações específicas, ferramentas e metodologias se torne importante. A ideia é produzir um documento de requisitos adequado, para daí obter a arquitectura e o desenho do sistema, e ainda permitir aos programadores começar a implementar o software.

Existem vários paradigmas na Engenharia de Requisitos, como por exemplo as orientadas a pontos de vista (do inglês, *viewpoints*) [Finkelstein e Sommerville, 1996], a casos de uso (do inglês, *use cases*) [Jacobson, 2003] e a objectivos (do inglês, *goals*) [Lamsweerde, 2001]. Esta dissertação foca o paradigma da Engenharia de Requisitos orientada a objectivos.

1.2 – Engenharia de Requisitos Orientada a Objectivos e Sistemas Multi-Agente (SMA)

Modelos organizacionais representam os objectivos de indivíduos, grupos ou organizações. Um objectivo é um propósito que o sistema deve alcançar, normalmente após o decorrer de uma acção (processo) [Kavakli, 2004]. Os objectivos dos intervenientes no sistema (do inglês, *stakeholders*), bem como o seu papel na tomada de decisões de desenho, são tópicos importantes na área da Engenharia de Requisitos. As abordagens orientadas a objectivos usam o conceito chave de objectivo, numa tentativa de melhor perceber e descrever aspectos no mundo real, de forma a lidar com a complexidade dos assuntos humanos. Pesquisas no campo dos planos de negócio, administração, controlo e análise de fluxos, apontam para uma abordagem orientada a objectivos. Esta abordagem baseia-se na premissa de que, em situações de trabalho cooperativo, as pessoas não seguem apenas regras e procedimentos. Elas procuram também ter conhecimento dos objectivos, individuais e do grupo, e agem de acordo com isso. Este aspecto é ainda mais evidente quando as pessoas não estão perante um processo repetitivo e bem estruturado [Kavakli, 2004].

Os objectivos são considerados uma possível motivação para uma acção. As abordagens baseadas em objectivos focam nas razões pelas quais um sistema é construído. Além disso, como os objectivos são geralmente mais estáveis que os requisitos, estas são uma fonte benéfica para derivação de requisitos [Chung et al., 1995]. Os objectivos são refinados em requisitos, podendo apontar para novos cenários e, reciprocamente, os cenários podem ajudar na descoberta de novos objectivos. Os objectivos relacionam-se com requisitos funcionais e não funcionais.

Os requisitos funcionais expressam o que um sistema deve ser capaz de fazer. Podem ser tarefas, serviços ou funções que o sistema deve estar apto a desempenhar para oferecer alguma funcionalidade aos seus utilizadores. São também conhecidos como requisitos operacionais ou comportamentais [Loucopoulos e Karakostas, 1995]. Durante o desenvolvimento do software, estes requisitos são incorporados em artefactos de software. No final do ciclo de desenvolvimento de software, todos os requisitos funcionais devem estar implementados de tal forma que o sistema satisfaça os requisitos definidos nos passos anteriores do processo. Este tipo de requisitos normalmente têm efeitos localizados, ou seja, apenas afectam uma porção do software que pretende satisfazer o requisito em questão [Loucopoulos e Karakostas, 1995].

Os requisitos não-funcionais (RNF) [Chung et al., 1995] estão relacionados com os atributos de qualidade dum sistema, e.g. desempenho, fiabilidade, segurança, manutenção, portabilidade, velocidade de resposta, disponibilidade, precisão, robustez, tratamento de erros, capacidade. São assim propriedades, ou qualidades, que um sistema possui. Eles deveriam ser sempre tidos em linha de conta mas, na maioria das vezes, não podem ser satisfeitos na sua totalidade. Alguns deles são mesmo conflituantes entre si (por exemplo, segurança e tempo de resposta) e impõem alguns compromissos na hora de implementar um sistema.

Estes requisitos são importantes na definição da qualidade do software e na justificação das opções de desenho. Definem o quão bem alguns aspectos estruturais e comportamentais devem ser satisfeitos. Estes requisitos devem ser determinados o mais cedo possível, dado que influenciam decisões tomadas ao nível do desenho e da implementação do sistema [Chung e Yu, 1998] [Chung, 1998] [Gross e Yu, 2001].

Uma estratégia para o desenvolvimento de software é recorrer a uma abordagem para Sistemas Multi-Agentes (SMA). Um SMA pode ser definido como uma rede fracamente acoplada de entidades capazes de resolver problemas e que, na impossibilidade de os solucionar sozinhos (são limitados), trabalham em conjunto para os resolver.

A *framework* i* foi desenvolvida com a finalidade de modelar ambientes organizacionais e os seus sistemas de informação. Os seus componentes procuram descrever relações de dependência entre vários actores num contexto organizacional bem como descrever os interesses dos *stakeholders*. Por *stakeholder* entende-se uma pessoa, ou organização, que de alguma forma é afectada pelo sistema [Kotonya e Sommerville, 1997]. Oferece uma abordagem à engenharia de requisitos baseada em agentes na medida em que é uma abordagem que se centra nos *stakeholders* do sistema e nas suas relações. Estas relações mostram como os actores dependem uns dos outros para alcançarem os seus objectivos. Esta é uma das razões que fazem do i* uma das abordagens escolhidas para desenvolver sistemas multi-agentes numa *framework* chamada Tropos [Castro et al., 2002].

Os SMA baseiam-se no comportamento organizacional dos agentes. A forma como estes se dispõem dentro de uma estrutura organizada influencia a forma como eles comunicam e interagem. Neste aspecto, é importante a definição de um estilo arquitectural organizacional, que permite a definição da arquitectura global do sistema em termos de subsistemas (actores), trocas de dados e fluxos de controlo

(dependências). A razão pela qual estamos a considerar a *framework* i* é porque esta tem sido utilizada para representar arquitecturas organizacionais.

1.1.3 – Arquitecturas e MDA

A arquitectura de software é uma das áreas em expansão dentro da disciplina de Engenharia de Software. Ganhou importância na última década e considera-se que exerça um papel importante em lidar com o desenvolvimento de software de grande porte. Tendo um papel organizativo, representam um suporte para desenvolver e manter sistemas capazes de resistir à mudança.

Não há uma definição unânime para arquitectura. Assim, a sua definição pode variar conforme o contexto de aplicação. No âmbito desta dissertação, podemos definir arquitectura como sendo um conjunto de "Questões estruturais que incluem a organização geral e estrutura global de controlo; protocolos para comunicação, sincronização e acesso a dados; tarefa de funcionalidade para projectar elementos; distribuição física; composição de elementos de projecto; escala e desempenho; e selecção entre alternativas de projecto" [Shaw e Garlan, 1996].

Tendo sempre por base a noção de arquitectura, o MDA (do inglês, *Model Driven Architecture*) define uma abordagem à especificação de sistemas de informação que separa a especificação da funcionalidade do sistema, da especificação da implementação dessa funcionalidade numa plataforma tecnológica em particular [Kleppe et al., 2007]. Esta abordagem, e as normas que a regem, permitem que um mesmo modelo, descrevendo a funcionalidade do sistema, possa ser implementado em diversas plataformas. Este factor torna-se importante à medida que as plataformas tecnológicas aparecem e se tornam ultrapassadas ou obsoletas. Pode-se dizer que o MDA foca sobre arquitectura, artefactos e modelos, na tentativa de aproveitar a utilidade destes últimos como ferramentas para a abstracção [Kleppe et al., 2007].

Tradicionalmente, as transformações entre modelos, ou de modelos para código eram feitas informalmente. Ao invés disso, em MDA, as transformações são sempre feitas por ferramentas. Este aspecto permite poupar muito esforço no desenvolvimento de software. Estas ferramentas pegam em modelos independentes da plataforma, transformam-nos em modelos específicos de uma plataforma e, posteriormente, em código. Uma transformação é constituída de uma definição de uma regra. É essa definição que indica a forma como um modelo é transformado noutro. Pode ser vista como uma colecção de regras de transformação que especificam, de forma não ambígua,

que partes de um modelo são utilizadas para criar um outro modelo [Kleppe et al., 2007].

1.2 Motivação / Problema

A *framework* i* oferece uma abordagem para guiar o desenvolvimento de sistemas multi-agentes (SMA). No entanto, o uso da notação i* como uma linguagem de descrição arquitectural (LDA) não é adequado, uma vez que esta notação apresenta algumas limitações na captura da informação necessária para projectar a arquitectura de SMA. Tendo por base o reconhecimento de que o UML 2.0 (do ingles, *Unified Modeling Language*) suporta a descrição arquitectural de software, esta dissertação automatiza a abordagem definida em [Silva C., 2003], que permite a especificação de transformação de modelos arquitecturais descritos em i* para modelos arquitecturais descritos num perfil de Agência UML definido em [Silva C., 2007], seguindo uma abordagem orientada a modelos (*Model Driven Engineering* – MDE) [Kleppe et al., 2007]. O metamodelo de agência é uma representação UML das propriedades que os agentes quando envolvidos num SMA. Esta automatização permitirá tornar mais célere o processo de mapear os conceitos presentes nos diagramas do i* para os diagramas que compõem o perfil de Agência. No caso concreto desta dissertação, procura-se detalhar e automatizar o mapeamento entre o diagrama de dependência estratégica do i* para um diagrama arquitectural do UML.

1.3 Objectivos e Contribuição

Esta dissertação de mestrado foi desenvolvida no contexto do projecto CAPES/GRICES (Proc. 129/05) "Integração de Técnicas de Requisitos com Aspectos: O Caso TROPOS", cujo objectivo é a elaboração de metodologias para desenvolver sistemas multi-agentes e a sua integração com a orientação a aspectos.

O objectivo desta dissertação é transformar modelos arquitecturais descritos em i*, mais concretamente o modelo de dependência estratégica, para modelos arquitecturais descritos num perfil (do inglês, *profile*) UML 2.0 de uma forma automatizada. Utilizaremos uma abordagem orientada a modelos para o efeito.

Existem algumas heurísticas informais que auxiliam o mapeamento dos modelos i* para um perfil de Agência UML [Silva C. et al., 2006a]. Esta dissertação melhora este conjunto de heurísticas, no qual esta dissertação se baseia, para descrever as regras de transformação utilizando os mecanismos de MDE.

O uso de MDE oferece vantagens ao nível da produtividade, portabilidade, interoperabilidade e manutenção e documentação. A saber [Kleppe et al., 2007]:

- Produtividade – a maior parte do código pode ser gerado;
- Portabilidade – Transformações possíveis para diversas plataformas de desenvolvimento;
- Interoperabilidade – Geração de ligações, ou pontes (do inglês, *bridges*), entre os modelos;
- Manutenção e documentação – Modelos não abandonados (rastreabilidade 100%).

Conseguindo concluir este processo de transformação, conseguimos uma abordagem de desenvolvimento de software com as vantagens da MDE e com as capacidades de modelação do UML.

1.4 Organização da dissertação

Este documento está organizado em sete capítulos. O Capítulo 1 é a presente introdução. No Capítulo 2 procura-se descrever a *framework* i*, descrevendo os seus conceitos e modelos constituintes. Este capítulo apresenta também o projecto Tropos, baseado no i*, como *framework* de desenvolvimento de sistemas multi-agentes (SMA). No seguimento introduzem-se as arquitecturas de software SMA, bem como dos estilos arquitecturais organizacionais existentes. Descrevem-se depois de estilos organizacionais em UML e dá-se uma breve referência a outras abordagens orientadas a objectivos.

No Capítulo 3 foca o desenvolvimento de software orientado a modelos. Este capítulo trata os conceitos de modelos independentes da plataforma e modelos específicos da plataforma (do inglês, *Platform-Independent Model* e *Platform-Specific Model* (PIM e PSM)). O capítulo termina fazendo uma pequena introdução Às transformações de modelos e metamodelação.

No Capítulo 4 apresenta a descrição da contribuição desta dissertação, onde são descritas novas heurísticas, e definidas as regras de transformação essenciais ao processo de automatização.

No Capítulo 5, apresenta um caso de estudo para demonstrar a viabilidade da nossa proposta, mostrando que se trata, de facto, de uma abordagem aplicável e vantajosa no contexto deste projecto. Finalmente, o Capítulo 6 apresenta as conclusões do trabalho.

2 – Requisitos e Arquitecturas Orientadas a Objectivos com TROPOS

Este capítulo descreve a *framework* i*, o desenvolvimento e arquitecturas de SMA. Discutimos ainda os estilos arquitecturais organizacionais de SMA e em UML. Finalmente, apresentamos outras abordagens orientadas a objectivos.

2.1 – Framework i*

A *framework* i* foi desenvolvida para modelar e racionalizar acerca de ambientes organizacionais e os seus sistemas de informação [Yu, 1995]. Oferece uma abordagem à engenharia de requisitos baseada em agentes (*"agent based"*). O termo *"agent based"* indica que esta abordagem se centra nos *stakeholders* do sistema e nas suas relações. Estas relações mostram como os actores dependem uns dos outros para alcançarem os seus objectivos. O suporte explícito para a modelação e análise de dependências ente múltiplos actores permite introduzir alguma análise social na análise do sistema, bem como na fase de desenho. O propósito deste tipo de análise é descobrir os "porquês" dos requisitos: porque é que um requisito é de um tipo e não doutro, ou porque é que esse requisito é necessário. Este tipo de estudo, baseado em contextos sociais/organizacionais, permite não só perceber os requisitos do sistema, como também ajudam a prepará-lo nas mudanças futuras [Yu, 1997].

2.1.1 – Conceitos

O conceito central no i* é o de actor intencional [Yu, 1993]. Os actores dentro de uma organização são vistos como tendo propriedades intencionais tais como objectivos, crenças, preocupações, capacidades e compromissos [Yu, 1997]. Estes actores dependem uns dos outros para atingirem objectivos, realizar tarefas e disponibilizar recursos. Interagindo entre si, conseguem atingir objectivos difíceis de alcançar sozinhos. No entanto, isto cria alguma vulnerabilidade, quando um actor depende de um outro que não conseguiu cumprir com as suas obrigações. Os actores são um ponto estratégico no sentido em que se preocupam com as oportunidades e as vulnerabilidades, e procuram as reconfigurações do seu ambiente que melhor se ajustam aos seus interesses [Yu, 1993] [Yu, 1995].

Os elementos da *framework* i* são descritos de seguida [Yu, 1993] [Yu, 1995]:

1. **Actor** – É uma entidade activa que realiza acções que lhe permitam atingir objectivos. São representados por um círculo nos diagramas.
2. **Dependência** – Descreve uma relação intencional entre dois actores. Neste relacionamento, um dos actores depende do outro em algo. Esse algo é designado por um dependum, que define o tipo da dependência e descreve a natureza do acordo entre as duas entidades. Uma dependência representa-se na forma Actor 1 –> Dependum –> Actor 2
3. **Objectivo** – É uma condição, ou estado do mundo que os *stakeholders* gostariam de alcançar. Não é especificada a forma como o objectivo vai ser alcançado, o que permite considerar várias alternativas. São permitidas dependências entre objectivos para representar delegações de responsabilidade em satisfazer um determinado objectivo. Um objectivo representa-se por uma forma oval.
4. ***Softgoal*** – É uma condição, ou estado do mundo em que os *stakeholders* gostariam de viver. Em oposição ao conceito de objectivo, não existe um critério concreto para determinar se o *softgoal* foi alcançado logo, está sujeito a um julgamento subjectivo por parte do programador na avaliação da satisfatibilidade do *softgoal*. A representação de *softgoals* é feita através de "nuvens".
5. **Recurso** – É uma entidade, física ou não, cujo principal requisito não funcional é a disponibilidade. Dependências que envolvem um recurso implicam que o actor detentor o forneça ao actor que dele depende. Os recursos representam-se por rectângulos.
6. **Tarefa** – Especifica uma maneira específica de fazer algo. Pode ser vista como uma solução no sistema que se pretende desenvolver. Estas soluções disponibilizam operações, processos, representações de dados, estruturação, restrições e agentes para satisfazer as necessidades extraídas dos *objectivos* e *softgoals*. Dependências envolvendo tarefas, ocorrem quando um actor depende de um outro para realizar uma actividade. As tarefas são representadas por hexágonos.

2.1.2 – Modelos

O i* consiste em dois modelos [Yu, 1997], o modelo de Dependência Estratégica (do inglês, *Strategic Dependency Model* (SD)), e o modelo de Razão Estratégica *(*do inglês, *Strategic Rationale Model* (SR)). Do modelo SD fazem parte um conjunto de nós e ligações entre eles, onde os nós representam actores e as ligações as dependências entre eles. O modelo SR é usado como complemento do SD, e serve para descrever preocupações e motivações dos participantes no processo, permitir a acessibilidade das possíveis alternativas na definição do processo e pesquisar em mais detalhe as razões inerentes às dependências entre os actores.

Mais concretamente, a figura 2.1 mostra o modelo SD do exemplo da marcação de reuniões (que vai ser explicado em detalhe mais à frente). Este modelo pode ser visto como uma rede de dependências entre vários actores, que captura a motivação e a razão de ser das actividades. Estimula um conhecimento mais profundo dos processos de negócio, criando um foco nas dependências intencionais entre actores, indo além do conhecimento usual baseado em actividades e fluxos entre entidades. Ajuda a identificar o que está em jogo, para quem e quais os impactos adjacentes à falha de uma dependência.

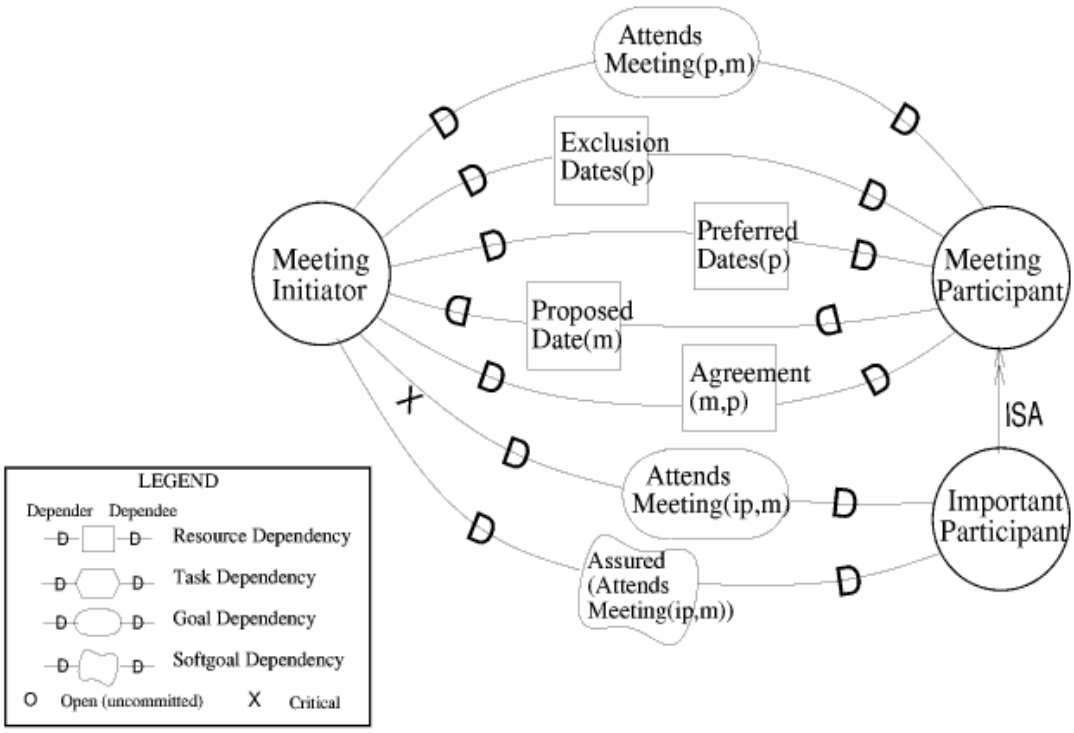

Figura 2.1 – Modelo de Dependência Estratégica para a marcação de reuniões sem a Agenda computorizada. [Yu, 1997]

O modelo de Razão Estratégica é suplementar ao modelo de Dependência Estratégica. Apesar deste oferecer dicas acerca da estruturação do processo, não é suficiente para sugerir, explorar e avaliar soluções alternativas. É este o papel do modelo SR.

Representando explicitamente relações de meio-fim (do inglês, *means-end*), o modelo SR apresentado na figura 2.2, que também se refere ao exemplo da marcação de reuniões, oferece uma forma sistemática de explorar o espaço de possibilidades no desenho do sistema. Este modelo permite capturar as razões associadas a cada actor e às suas dependências. Cada actor possui dependências exteriores com outros actores mas, internamente a cada um, existem objectivos, tarefas que justificam cada uma dessas dependências.

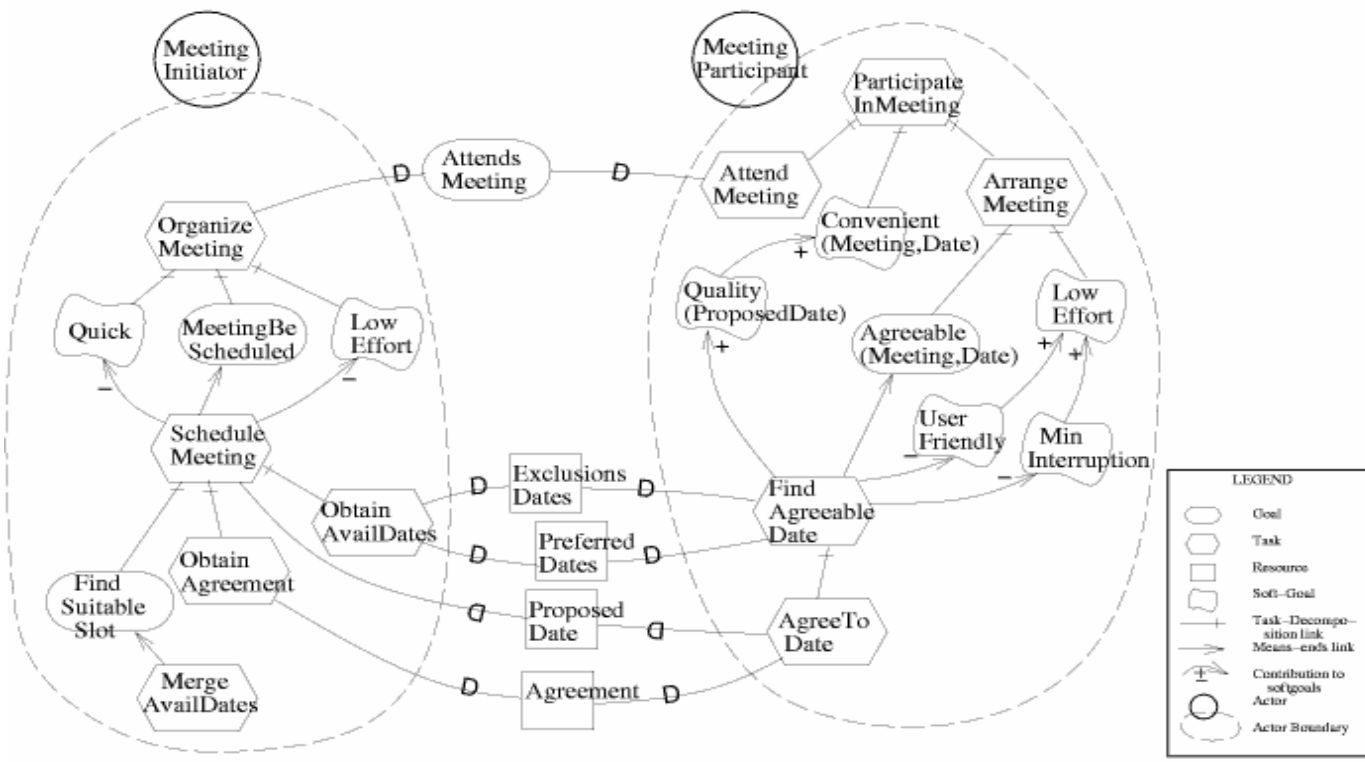

Figura 2.2 – Modelo de Razão Estratégica para a marcação de reuniões sem Agenda computorizada. [Yu, 1997]

Um modelo SR é um grafo com quatro tipos de nós: objectivo, tarefa, recurso e *softgoal* com as mesmas características descritas para o modelo SD. Para além destes nós, este modelo tem ainda dois tipos principais de ligações: ligação meio-fim e ligação de decomposição de tarefa (do inglês, *task decomposition link*) [Yu, 1997]:

- Ligação meio-fim – Ligações conotadas com a intenção de atingir um fim. Esse fim pode ser um objectivo, recurso, *softgoal* ou tarefa. Os meios para obter este fim são definidos como tarefas que são necessárias para atingir este fim. Um tipo especial de ligação meio-fim é o de contribuição. Uma relação de contribuição indica os objectivos ou as tarefas que contribuem positiva ou negativamente para a concretização dum objectivo em particular.
- Ligação de decomposição de tarefa – Um nó tarefa é ligado às suas tarefas componentes através de uma ligação de decomposição. Os quatro tipos de nós podem ser ligados por esta ligação. Uma tarefa pode ser decomposta em outras mais pequenas de forma a representar melhor as razões associadas ao seu cumprimento.

Quando um elemento é expresso como um objectivo no modelo SR, significa que este pode ser alcançado de várias maneiras. Uma tarefa representa uma forma de o fazer. Sempre que é necessário redesenhar um processo de negócio, os objectivos são o melhor sítio para começar tendo em vista o melhoramento do sistema. Nesta fase importa perguntar "de que outra maneira podemos alcançar os objectivos?". Cada alternativa terá impactos diferentes nos *softgoals*, nomeadamente em termos de disponibilidade, performance e segurança. Apesar de não haver critérios concretos para medir se um *softgoal* é satisfeito ou não, é sempre possível avaliar qual das alternativas oferece uma contribuição mais positiva para a generalidade destes objectivos qualitativos. Cada maneira de atingir um objectivo tem contribuições positivas e negativas para os *softgoals*. Compete a quem vai implementar o sistema decidir qual a melhor alternativa.

Normalmente, quando se tenta perceber uma organização, a informação capturada pelas técnicas de modelação consideradas de padrão (Diagramas de Fluxo de Dados, Diagramas de Entidades e Relações, Máquinas de estados, etc.), o foco encontra-se nas entidades, funções, fluxos de dados e estados. Estas técnicas não são capazes de expressar razões para o processo em termos de motivações e intenções. É neste aspecto que os modelos do i* ganham vantagem. Nos casos em que é necessário melhorar os processos de negócio de uma organização, uma vez que esta tarefa é simplificada quando se descrevem os quês e os porquês de um sistema [Yu, 1997].

2.1.3 – O processo

O processo do i* começa por identificar os actores (ou agentes) envolvidos no processo ou sistema a ser modelado. Estes podem ser identificados considerando quem ou o quê, no sistema, tem a característica de intencionalidade. É apropriado o analista dar intencionalidade a agentes não humanos, se isto se revelar vantajoso para o processo de análise. Uma vez identificados os agentes, é construído o modelo SD representando os objectivos de alto nível e as dependências doutros agentes [Chitchyan et al., 2005].

Os diagramas do modelo SD são então refinados no modelo SR. Este refinamento faz-se olhando para dentro de cada agente. Aqui importa analisar a forma como cada agente avalia os seus objectivos internamente.

Estudando os diagramas dos modelos SD e SR, o analista pode derivar várias estruturas de dependências alternativas e outras formas de atingir os objectivos pretendidos, ou seja, novos modelos SD e SR. Cada uma destas alternativas tem de ser testada contra vulnerabilidades. O i* sugere que, se existe um ciclo de dependências (dois agentes dependem um do outro, directa ou indirectamente), é natural que o objecto da dependência, o dependum, influencie a escolha dessa estrutura. De entre as várias alternativas, selecciona-se aquela que aparente ser a melhor.

2.1.4 – Exemplo prático

Nada melhor que um exemplo para ilustrar a utilidade e as potencialidades do i* e de cada um dos seus modelos. Nesta situação particular, considere-se uma agenda de reuniões computorizada, capaz de fazer marcações. Os requisitos dizem que, para cada pedido de reuniões, a agenda deverá determinar uma data e um local que satisfaça as possibilidades da maioria dos participantes no evento. O "*Meeting Initiator*" deverá recolher informação de todos os potenciais participantes acerca da sua disponibilidade de se reunirem num determinado intervalo de datas, baseada nas suas agendas pessoais. Desta informação faz parte um conjunto de exclusão (datas em que o participante não pode ir) e um conjunto de preferências (datas preferidas pelo participante). A Agenda propõe uma data. Essa data não pode pertencer ao conjunto de exclusão, e deverá pertencer ao maior número possível de conjuntos de preferências. Os participantes acordam ir a uma reunião assim que uma data aceitável seja encontrada.

Muitas abordagens e técnicas de engenharia de requisitos têm sido desenvolvidas para ajudar a refinar este tipo de requisitos, de forma a atingir uma melhor precisão, completude e consistência. No entanto, para desenvolver sistemas que

vão de encontro ao que uma organização verdadeiramente precisa, é necessário um conhecimento profundo de como o sistema vai ser incorporado no ambiente organizacional. Por exemplo, o engenheiro de requisitos, antes de refinar os requisitos iniciais pode, e deve, fazer as seguintes perguntas:

- Porque é necessário marcar reuniões com antecedência?
- Porque é que o "*Meeting Initiator*" precisa perguntar aos participantes por datas de exclusão e de preferência?
- Porque será desejável ter uma agenda de reuniões computorizada? Com interesse para quem?
- Uma confirmação pelo sistema computorizado é suficiente? Se não, porque não?
- Há participantes mais importantes que outros? Se sim, porquê?

A maioria dos modelos de requisitos não estão bem preparados para responder a estas questões. Tendem em focar os quês em vez dos porquês. As respostas a estas questões são importantes na medida em que ajudam na implementação do sistema, na interligação com outros componentes com os quais o sistema poderá ter de cooperar e até na evolução futura do sistema.

Para um conhecimento mais profundo acerca de como a Agenda de reuniões se vai incorporar no ambiente organizacional, o modelo SD foca as relações intencionais entre os vários actores organizacionais. Realçando as dependências que os actores têm entre si, ajuda a uma percepção melhor dos porquês.

Olhando primeiramente para a configuração organizacional, no modelo SD da figura 2.1 pode-se ver que o "*Meeting Initiator*" depende da disponibilidade dos participantes para participar na reunião. Se algum participante não comparece a uma reunião, este pode não conseguir cumprir com os objectivos a que se propõe. Esta é a razão para se querer marcar uma reunião com antecedência. Assim, o "*Meeting Initiator*" precisa que os participantes forneçam informação acerca da sua disponibilidade, em termos das suas preferências. Para chegar a um acordo, os participantes precisam que o "*Meeting Initiator*" lhes envie as datas propostas. A seguir o "*Meeting Initiator*" depende dos participantes concordem com a data. Relativamente aos participantes importantes, o "*Meeting Initiator*" depende de forma crítica (denotada por um X na notação gráfica) da sua disponibilidade para comparecer na reunião e, além disso, da sua confirmação de que realmente conseguem assistir ao evento.

Os vários tipos de dependências são usados para diferenciar os tipos de relações entre dois actores. A dependência do "*Meeting Initiator*" relativamente à disponibilidade do participante é uma dependência de objectivo. É da responsabilidade do participante obter esse objectivo. O acordo acerca da data da reunião é modelado como uma dependência de recurso. Significa isto que se espera que o participante apenas dê um acordo relativamente às datas. Se não existir acordo, é o "*Meeting Initiator*" que tem de encontrar outras datas para ultrapassar o problema. No caso dos participantes importantes, o "*Meeting Initiator*" depende de uma forma crítica da presença dessas pessoas. Pretende-se que haja uma confirmação posterior por parte dos participantes. Isto é modelado como sendo uma dependência de *softgoal* e é quem depende que decide qual a medida que lhe é suficiente, por exemplo, uma confirmação telefónica. Estes tipos de dependência não podem ser expressados em modelos não intencionais.

A figura 2.3 mostra um modelo SD da marcação de reuniões recorrendo a uma agenda computorizada. O "*Meeting Initiator*" delega muito do trabalho na Agenda e assim já não precisa de se preocupar em obter informação sobre a disponibilidade dos participantes. Não é importante para ele como é que a Agenda faz isto, desde que sejam encontradas datas aceitáveis. Esta relação é apresentada como uma dependência de objectivo do "*Meeting Initiator*" para a Agenda. A Agenda espera que o "*Meeting Initiator*" indique o intervalo de datas através de um procedimento específico. Isto é modelado através duma dependência de tarefa.

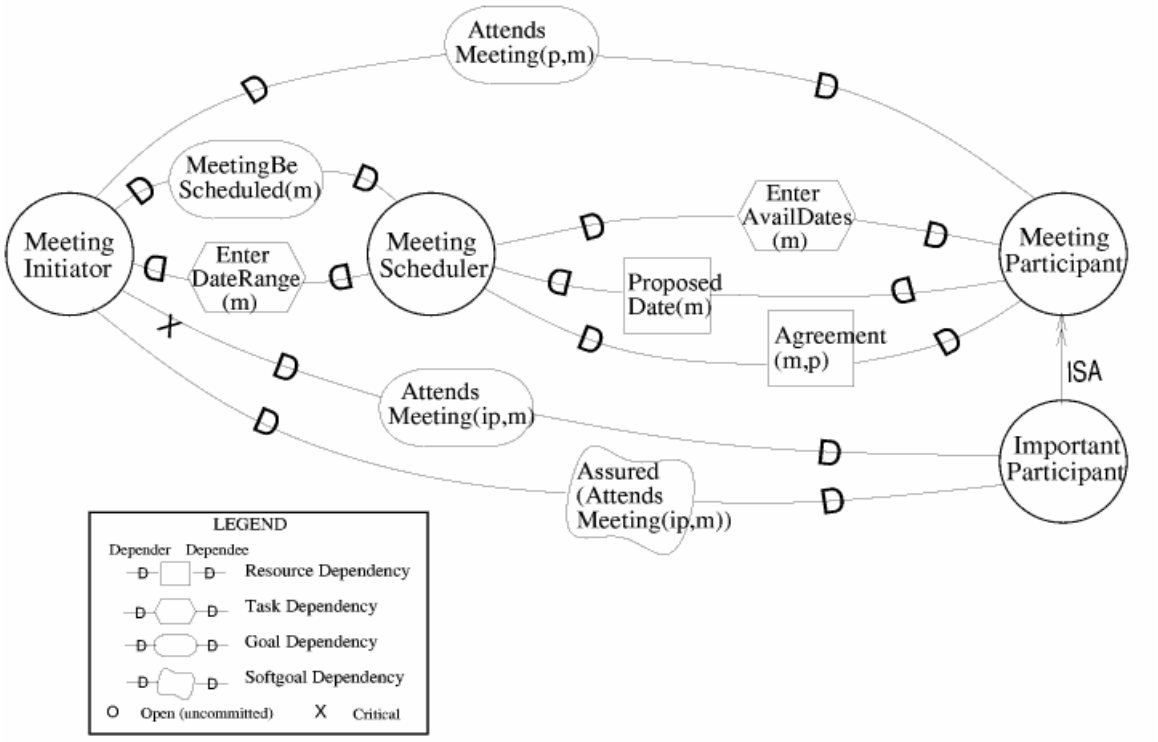

Figura 2.3 – Modelo de Dependência Estratégica para a marcação de reuniões com Agenda computorizada. [Yu, 1997]

De notar que o "*Meeting Initiator*" ainda depende da disponibilidade dos participantes em comparecer na reunião. É ele, e não a Agenda, que tem interesse em que os participantes vão à reunião. Reunir a garantia de que os utilizadores importantes vão, de facto à reunião, não é delegada à Agenda, mantém-se antes como uma dependência do "*Meeting Initiator*" para os participantes importantes.

Este segundo modelo SD permite uma análise em termos de oportunidades e vulnerabilidades. Como oportunidade considera-se a capacidade que a Agenda tem em garantir o objectivo *MeetingBeScheduled* tirando esse peso do "*Meeting Initiator*". Por outro lado, o "*Meeting Initiator*" torna-se vulnerável a uma falha na Agenda no cumprimento da sua função.

O modelo SD oferece um nível de abstracção na descrição de ambientes organizacionais e dos seus sistemas de informação. Mostra relações externas entre actores, escondendo as construções intencionais existentes dentro de cada actor. Como já foi dito, este modelo pode ser útil para perceber a configuração organizacional mas é também útil ter uma representação mais explícita dos actores.

É aqui que entra o modelo de Razão Estratégica, que oferece uma modelação mais detalhada olhando para dentro dos actores e modelando as suas relações

intencionais internas. Elementos intencionais como objectivos, tarefas, recursos e *softgoals*, aparecem no modelo SR não apenas como dependências externas, mas também como elementos internos ligados por ligações de meio-fim e ligações de decomposição de tarefas. O modelo SR da figura 2.2 trabalha sobre a relação entre o "*Meeting Initiator*" e um participante como descrito no modelo SD da figura 2.1.

Por exemplo, para o "*Meeting Initiator*", um objectivo interno será o "*MeetingBeScheduled*". Este objectivo pode ser alcançado (representado por uma ligação de meio-fim) marcando reuniões de uma certa maneira, que consiste em (representado por uma ligação de decomposição de tarefa) obter as disponibilidades dos participantes, encontrar a melhor data e obter a concordância dos participantes.

Estes elementos da tarefa "*ScheduleMeeting*" são representados como subobjectivos, subtarefas ou recursos dependendo do tipo de liberdade de escolha dada para a decisão de como alcançar os objectivos. Além disso, o facto do objectivo "*FindSuitaleSlot*" ser um subobjectivo, indica que pode ser alcançado da mais do que uma maneira diferente. Já as tarefas "*ObtainAvailDates*" e "*ObtainAgreement*", referem-se a maneiras específicas de realizar essas tarefas. Da mesma forma, o facto de *"MeetingBeScheduled"* ser representado como objectivo, indica que o "*Meeting Initiator*" acredita que há mais do que uma maneira de alcançar o objectivo.

O "*MeetingBeScheduled*" é, ele próprio, um elemento da tarefa de mais alto nível que é organizar uma reunião. Outros subobjectivos inerentes a esta tarefa podem incluir a encomenda de equipamento, ou o envio de memorandos. Esta tarefa tem dois elementos adicionais, especificando que a marcação de uma reunião deve ser rápida e feita sem muito esforço. Estes critérios qualitativos são modelados como *softgoals*. São usados para avaliar novos meios para atingir o objectivo. Nesta figura 2.2, temos que a marcação de reuniões contribui negativamente para os *softgoals* "*quick*" e "*LowEffort*".

Do lado dos participantes, espera-se que façam a parte deles no processo da marcação de uma reunião. Para isto, basta ao participante dar a sua informação de disponibilidade ao "*Meeting Initiator*" e depois concordar com uma das datas propostas. Os participantes querem que as datas das reuniões sejam convenientes e que não impliquem muitas interrupções.

O modelo SR oferece uma forma de modelar os interesses dos *stakeholders*, e de como estes podem ser satisfeitos. Analisando a figura 2.2, pode-se ver que o "*Meeting Initiator*" não está satisfeito com o esforço necessário para marcar uma reunião, bem como com o tempo que esta tarefa demora. Assim, pode delegar o subobjectivo de

marcar uma reunião a uma Agenda computorizada. Na figura 2.4 podemos ver um modelo SR modelando a possibilidade de inserir uma Agenda computorizada, capaz de resolver a questão da marcação de reuniões.

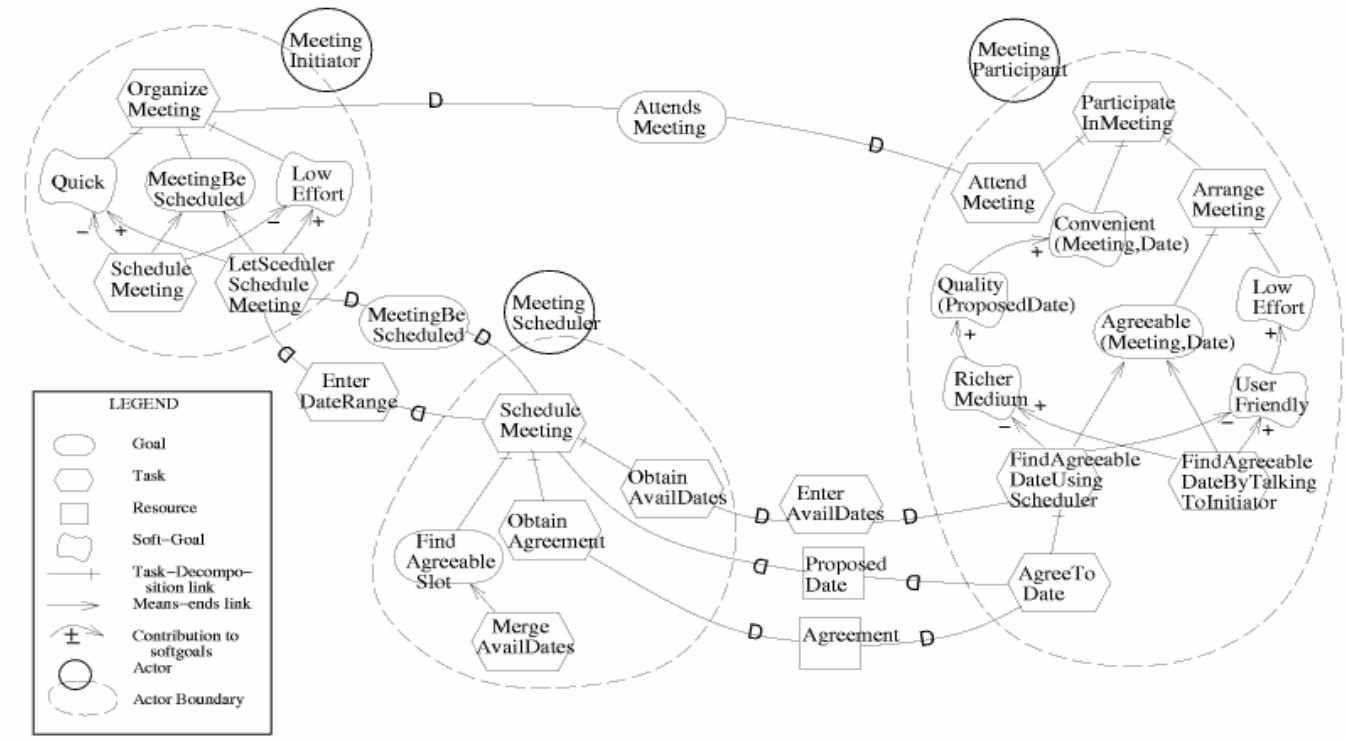

Figura 2.4 – Modelo de Razão Estratégica para a marcação de reuniões com Agenda computorizada. [Yu, 1997]

No entanto, o uso deste sistema requer que os participantes insiram a sua informação de disponibilidade num determinado formato. Isto é modelado como uma dependência de tarefa nos participantes. Desta forma, esta nova solução também se pode revelar negativa em termos de facilidade de uso, o que afecta directamente o esforço necessário por parte dos participantes para dar conta da sua disponibilidade ao sistema. Todas as alternativas terão de ser sujeitas a um estudo de funcionalidade e de viabilidade e, depois de todas as alternativas terem sido avaliadas, proceder-se-à à escolha de qual delas serve melhor aos interesses da organização e dos seus colaboradores.

2.2. – Uma framework de Desenvolvimento de SMA: TROPOS

O projecto Tropos [Castro et al., 2002] propõe uma metodologia de desenvolvimento de software baseada nos conceitos usados para modelar requisitos iniciais. Em particular, esta proposta adopta o i* de Eric Yu, que oferece as noções de

actor, objectivo e de dependência [Yu, 1997], usando-as como base para modelar os requisitos iniciais e finais, desenho (ou projecto) arquitectural e detalhado e implementação (as 5 fases da metodologia Tropos) [Silva C., 2003].

Nesta abordagem, a análise inicial de requisitos assume um papel fundamental, precedendo a especificação do sistema. Este aspecto pode ser visto como uma inovação já que até à data pouca atenção tem sido dada às actividades precedem a formulação dos requisitos iniciais. De entre as actividades desta fase inicial na engenharia de requisitos, destacamos aquelas que se relacionam com a forma como o sistema conseguirá responder aos propósitos organizacionais, porque é que o sistema é necessário, que alternativas podem existir, quais as implicações dessas alternativas para os interesses dos vários *stakeholders*. Aqui a intenção é compreender os "porquês" [Yu, 1993] dos requisitos do sistema, mais do que determinar o que o sistema tem de fazer. É no âmbito este processo inicial, que a *framework* do i* ganha força, por possuir características de modelação e capacidades de raciocínio que podem ser consideradas apropriadas para esta fase da engenharia de requisitos.

Esta primeira fase deste processo revela-se bastante importante pelos seguintes motivos:

- O desenvolvimento de sistemas baseia-se muito em suposições acerca do domínio das tarefas. Já foi estudado em [Curtis et al., 1988] que um fraco conhecimento do domínio é uma das causas primeiras de falhas num projecto. Ter um bom conhecimento acerca do domínio implica perceber os interesses e prioridades dos vários actores em jogo. Além disso importa ter uma boa compreensão dos conceitos associados a esse domínio.
- Os utilizadores precisam de ajuda aquando da elicitação inicial de requisitos. Torna-se necessária uma *framework* sistemática para ajudar os membros da equipa de implementação a perceber o que os clientes querem, e para ajuda-los a perceber o que o sistema pode fazer por eles [Grudin, 1988].
- Os utilizadores dos sistemas estão cada vez mais inseridos na reestruturação dos processos de negócio. Os sistemas são vistos como um meio para soluções de negócio inovadoras [Hammer e Champy, 1993]. É cada vez mais importante relacionar os objectivos do sistema com os objectivos do negócio e organizacionais.

- Lidar com a mudança é um dos maiores desafios da engenharia do software actualmente. Ter uma representação do que é importante dentro de uma organização em modelos e requisitos, poderá permitir que as mudanças no software possam ser seguidas desde a sua fonte (as alterações organizacionais que levaram à alteração dos requisitos) [Gotel e Finkelstein, 1994].
- Ter uma base sólida em termos organizacionais, e de conhecimento estratégico, irá permitir que esse conhecimento seja partilhado em vários domínios de aplicação. Isto iria também facilitar a reutilização de software nesses mesmos domínios.
- Os sistemas de hoje em dia interagem muito entre si, e é cada vez mais importante determinar a forma como esses sistemas cooperam para alcançarem os objectivos de uma empresa. Os modelos de requisitos na fase inicial que lidam com os objectivos organizacionais e com os interesses dos seus *stakeholders* são transversais a vários sistemas e permitem uma visão da cooperação entre sistemas dentro de um contexto organizacional.

As actividades conotadas com as actividades feitas nesta fase inicial foram sempre feitas de forma informal e sem muito suporte ao nível de ferramentas. À medida que a complexidade no domínio dos problemas vai aumentando, torna-se evidente que há a necessidade de usar algum tipo de instrumentos de forma a facilitar a tarefa do engenheiro de requisitos. Toda uma base de conhecimentos pode ser obtida durante a fase inicial, que pode depois ser usada para racionalizar os objectivos organizacionais, as alternativas ao sistema, as implicações para os *stakeholders*, etc. É importante manter esta base de conhecimentos de forma a guiar o desenvolvimento do sistema, bem como ajudar a lidar com as mudanças ao longo do ciclo de vida de uma aplicação.

Existem várias *frameworks* para representar o conhecimento em engenharia de requisitos [Bubenko, 1993] [Dardenne et al., 1993] [Lamsweerde et al., 1995]. No entanto estas não distinguem fase inicial da fase final em engenharia de requisitos. Mais do que isso, a maioria das técnicas de requisitos existentes estão talhadas para a fase final da engenharia de requisitos que foca os conceitos de consistência, *"completude"* e a verificação automática de requisitos. Em oposição, a análise inicial de requisitos procura modelar os interesses dos *stakeholders* e como satisfazê-los com as várias alternativas de sistemas. No final desta fase temos os modelos de dependência e razão estratégicas [Silva C., 2003].

Na análise final de requisitos o modelo conceptual é estendido para incluir um novo actor, que representa o sistema e as suas dependências com outros actores do ambiente. Estas dependências definem os requisitos funcionais e não-funcionais do sistema a ser construído. O propósito nesta fase é produzir um documento de requisitos para passar às equipas de desenvolvimento, de modo a que o sistema daí resultante esteja adequadamente especificado.

A arquitetura do sistema descreve o sistema de software a um nível macroscópico em termos do número de subsistemas, componentes e módulos inter-relacionados através de dependências de dados e de controlo. Assim, a intenção do Desenho Arquitectural é a definição da arquitectura global do sistema em termos de subsistemas (actores), trocas de dados e fluxos de controlo (dependências). O Desenho Arquitectural está dividido em três etapas: Escolher o estilo arquitectural usando como critério as qualidades desejadas, que foram identificadas na fase anterior e conduzir uma análise efectiva desses atributos de qualidade particulares do software. Incluir novos actores, de acordo com a escolha do estilo arquitectural. Numa segunda fase, incluir novos actores e dependências, assim como decompor os actores e as suas dependências em sub-actores e sub-dependências. Rever os modelos de razão e dependência estratégicas [Silva C., 2003]. E, além disso, definir as capacidades (habilidades) dos actores a partir das tarefas que eles próprios e os seus sub-actores terão que realizar para atingir por completo os requisitos funcionais e satisfazer os não-funcionais. Por fim, definir um conjunto de tipos de agentes (componentes), e atribuir a cada componente uma ou mais habilidades diferentes de modo a solucionar todos os objectivos definidos ao nível arquitectural. É nesta fase de Desenho Arquitectural que esta dissertação se encaixa.

Na fase de Desenho Detalhado normalmente a plataforma de desenvolvimento já foi escolhida, para permitir que o resultado desta fase seja mapeado directamente para o código. Aqui a ideia é especificar as capacidades dos agentes e as suas interacções. Cada agente é definido mais especificamente em termos de entradas, saídas, controlo e outras informações relevantes.

Chegados à fase de implementação, estabelece-se o mapeamento entre a plataforma de implementação e o Desenho Detalhado e segue-se passo a passo a especificação deste último.

2.3 – Arquitectura de Software SMA

No âmbito da fase de Desenho Arquitectural do Tropos, onde a arquitectura global do sistema é definida, importa realçar que um estilo arquitectural específico deve ser escolhido tendo como critério as qualidades identificadas na fase da análise final de requisitos.

Os sistemas multi-agentes estruturam o sistema enquanto conjunto de elementos que cooperam entre si para realizarem os requisitos do sistema. Além disso ajudam a compreender as propriedades externamente visíveis, bem como as relações entre eles.

Os elementos típicos da arquitectura de um sistema multi-agente são os agentes, o ambiente, recursos, serviços, etc. As relações entre estes elementos são muito diversas e vão desde interacções, através do seu ambiente, entre agente cooperativos, até à negociação de complexos protocolos numa sociedade de agentes com interesses próprios. Resumindo, a abordagem multi-agente é flexível ao ponto de ser útil nos mais variados domínios de aplicação.

Dependendo do ambiente envolvente e do domínio de aplicação, um sistema baseado em agentes pode ter um ou mais agentes. Existem casos em que um único agente é suficiente. No entanto, do ponto de vista da Engenharia do Software, a modelação de sistemas com vários agentes interagindo entre si é geralmente mais interessante.

2.3.1 – Estilos Arquitecturais Organizacionais

As técnicas de desenvolvimento de software actuais geralmente levam a um software inflexível e não genérico. Actualmente o software tem de se basear em arquitecturas capazes de evoluir e suportar alterações sem custos elevados. Quer seja para acomodar novos componentes ou lidar com novos requisitos, estas modificações vão ocorrer mais tarde ou mais cedo. Torna-se necessário que as arquitecturas sejam capazes de acomodar estas alterações.

O conceito de arquitectura evoluiu, e é agora mais do que simplesmente estrutura, consegue agora também definir comportamentos. Isto vai de encontro às exigências actuais que pretendem arquitecturas flexíveis, com componentes fracamente acoplados de forma a suportar as modificações que arquitecturas altamente optimizadas para um conjunto inicial de requisitos.

Neste âmbito, o projecto Tropos [Castro et al., 2002] definiu um catálogo de estilos arquitecturais organizacionais para aplicações multi-agentes cooperativas,

dinâmicas e distribuídas [Silva C., 2003]. A criação destes estilos foi inspirada na teoria da organização [Hatch e Cunliffe, 2006]. De facto, uma organização pode ser vista como um compromisso social que persegue objectivos colectivos, que controla o seu próprio desempenho e que tem uma fronteira que a separa do seu ambiente. Assim, um sistema de software dentro de uma empresa pode ser analisado nos mesmos trâmites que uma organização. Sempre que pessoas interagem em organizações, existem muitos factores em jogo. Os estudos organizacionais procuram perceber e modelar estes factores.

Sistemas multi-agentes são promissores como modelos de organização, porque são baseados na ideia de que a maior parte do trabalho feito em organizações humanas se baseia em inteligência, comunicação, cooperação e processamento paralelo. Oferecem uma alternativa às teorias da organização que são de natureza abstracta e não dão atenção ao nível do agente. Contrariamente às teorias de organização clássicas, que são uma rica fonte de inspiração para o desenvolvimento de modelos multi-agentes pelo seu foco ser o nível de agente [Gazendam e Jorna, 1993]. Serve isto para realçar a importância da modelação organizacional, como um passo inicial crítico na produção de aplicações de negócio.

Estilos arquitecturais organizacionais incluem União estratégica (*joint venture*), estrutura em 5 (*structure in 5*), integração vertical (*vertical integration*), apropriação (*co-optation*), comprimento de braço (*arm's length*), tomada de controlo (*takeover*), pirâmide (*Pyramid*), contratação hierárquica (*hierarchical contracting*), oferta (*bidding*) e estrutura plana (*flat structure*) definidos em [Kolp et al., 2002]. Neste relatório apenas se apresentam os dois primeiros.

O estilo **União Estratégica *(Joint Venture)***. Este consiste num acordo entre dois ou mais actores, que permita alcançar o maior benefício possível, com o mínimo de custos relacionados com investimentos e manutenção. Um desses actores é o responsável pela gestão e coordenação de tarefas, operações e pela partilha de conhecimento e recursos. É de notar que todos os actores perseguem objectivos comuns. Esta gestão pode ser feita localmente ou globalmente. No primeiro caso, o actor principal interage directamente com outros actores principais para troca de dados e serviços. Numa dimensão global, o modo de funcionamento e a coordenação estratégica do sistema, e dos seus actores secundários, é feita pelo actor responsável comum. Nesta organização, os actores secundários fornecem os serviços e garantem a execução das tarefas necessárias para o bom funcionamento do sistema.

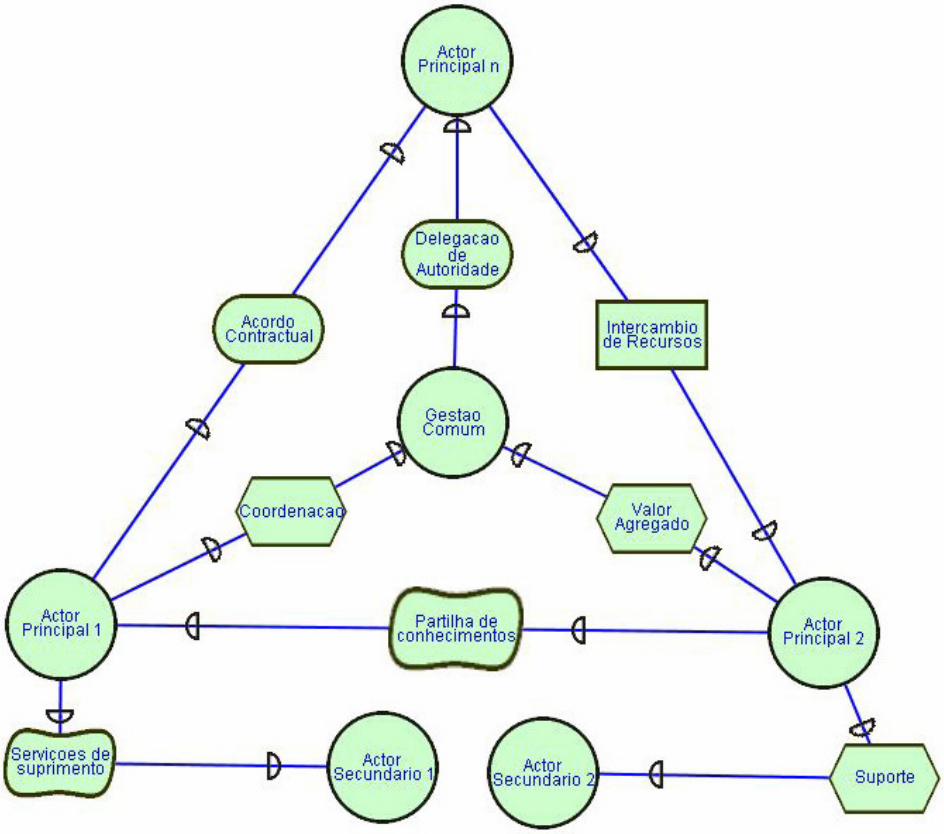

Figura 2.5 – União Estratégica (adaptado de [Castro et al., 2002])

Já o estilo **Estrutura em 5** ***(Structure in 5)*** consiste em componentes estratégicos e logísticos, que geralmente se encontram nas várias organizações. Ao nível da base, tem-se o *Núcleo Operacional*. É aqui que as tarefas e operações básicas, como os procedimentos de entrada, saída e de processamento no apoio ao funcionamento do sistema, são executados. No topo da organização encontra-se o componente *Cúpula*, composto de actores executivos estratégicos. Abaixo dela, situam-se os componentes de logística, controlo e gestão, respectivamente *Suporte*, *Coordenação* e *Agência Intermediária*. A componente de *Coordenação* trata de padronizar o comportamento das outras componentes para além de aplicar procedimentos analíticos para ajudar o sistema a adaptar-se ao ambiente em que vai ser inserido. A componente de *Suporte* dá assistência ao *Núcleo Operacional*, para serviços não operacionais, que estão fora do fluxo básico de tarefas e procedimentos operacionais. Actores que se juntam da *Cúpula* estratégica ao *Núcleo Operacional* caracterizam a Agência Intermediária.

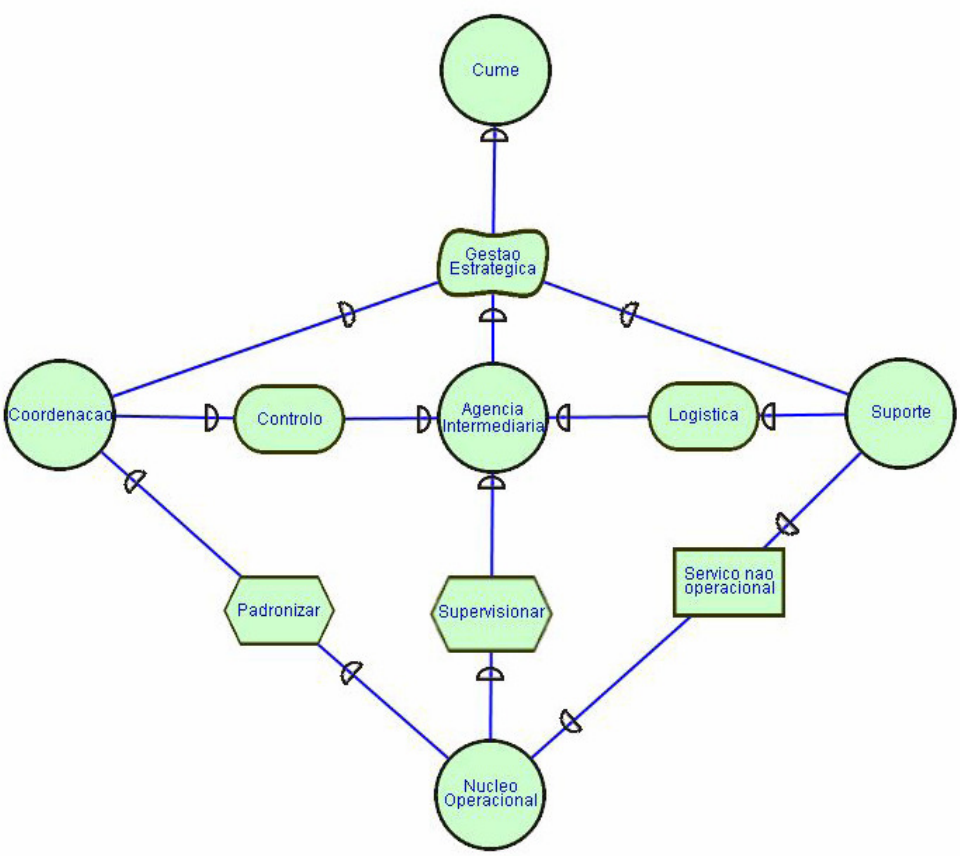

Figura 2.6 – Estrutura em 5 (adaptado de [Castro et al., 2002])

2.4 – Estilos Arquitecturais Organizacionais em UML

É do conhecimento da comunidade científica que o uso do i* como uma linguagem de descrição arquitectural é inadequado, dado que não consegue modelar algumas características arquitecturais de SMA tais como portas, conectores, protocolos e interfaces. Assim, na parte do i*, é necessário fazer uma análise meio-fim de cada actor pertencente à arquitectura SMA. Esta análise ajuda na identificação das razões, ou motivações, associadas a cada dependência que um actor possua. O objectivo principal do sistema é decomposto numa ou mais tarefas através de ligações de meio-fim. Algumas dessas tarefas têm elas também de ser decompostas. Essa decomposição consegue-se subdividindo a tarefa em uma ou mais tarefas, *softgoals* e recursos. Subtarefas já não podem ser decompostas dado que o elemento na relação de meio-fim só pode ser um objectivo ou um *softgoal* [Silva C. et al., 2006a]. A questão é, de alguma forma mapear modelos arquitecturais descritos em i* para modelos arquitecturais descritos de acordo com o metamodelo de Agência (secção 3.6.2). Para isso foram definidas algumas heurísticas para guiar o mapeamento dos conceitos que o i* oferece

para descrever arquitecturas SMA, para os conceitos definidos no metamodelo de Agência. Essas heurísticas serão discutidas no Capítulo 4.

O UML apresenta-se como uma ferramenta poderosa na modelação das propriedades dos sistemas multi-agente. O metamodelo de Agência foi concebido como um perfil da UML para permitir o seu uso na representação das propriedades que os agentes precisam de ter para se conseguir especificar um SMA.

Para permitir o uso de ferramentas UML na modelação de SMA, definiu-se em [Silva C., 2007] um perfil UML para o metamodelo de Agência. Este perfil possibilitou a extensão de algumas metaclasses UML de forma que alguns diagramas do UML fossem usados para modelar as seis vistas do desenho de SMA, a saber: Arquitectural, Comunicação, Ambiental, Intencional, Racional e Acção.

Esta dissertação irá concentrar-se apenas no diagrama arquitectural que é uma extensão ao diagrama de classes da UML, usado para modelar a visão estática dos módulos que compões um sistema [Silva C., 2007]. Para explicar este diagrama, usaremos como exemplo o padrão cliente-servidor [Shaw e Garlan, 1996]. O diagrama da Figura está definido em termos de *AgentRoles* que possuem *goals* que são conseguidos através de Planos. Um agente precisa de interagir com outros para corresponder às suas responsabilidades. Um *AgentRole* possui *OrganizationalPorts* que permitem a troca de mensagens entre agentes através de *AgentConnectors* para obtenção de um dependum.

É isso que ilustra a figura 2.7. O agente com o *AgentRole Client* procura alcançar o *goal ServiceRequest* executando o *MacroPlan RequestPlan* que, por sua vez, consiste em executar a *ComplexAction request().*

Um agente, como o papel de *"provider"*, é responsável por fazer o serviço definido pelo dependum. Este *AgentRole* procura levar a cabo o *goal ServicePerformed* executando o *MacroPlan PerformPlan*, que por sua vez consiste em executar a *ComplexAction service().*

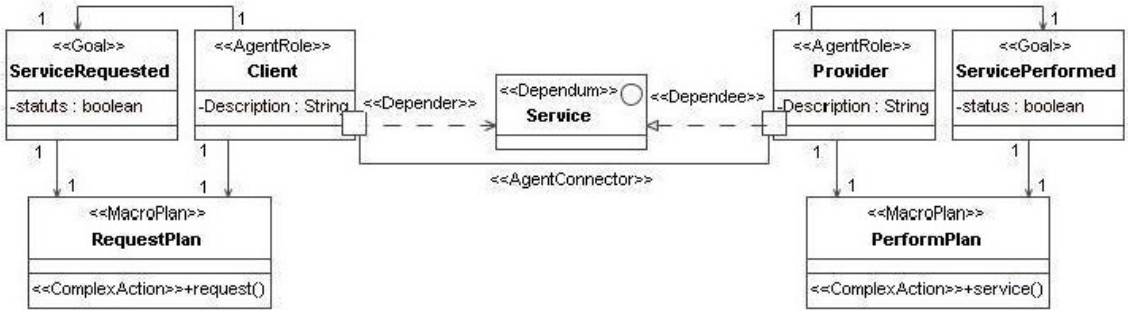

Figure 2.7 – SMA – Diagrama arquitectural [Silva C., 2007]

As trocas de mensagens que permitem esta partilha entre os agentes são feitas através de um AgentConnector. A informação referente ao tipo do dependum (*DependumKind*), grau (*DegreeKind*) e confiança (*TrustKind*), fazem parte da especificação do dependum [Silva C., 2007].

2.5 – Outras abordagens Goal Oriented (KAOS e v-graph)

2.5.1 – KAOS

Na linha do projecto Tropos, existem outras abordagens orientadas a objectivos. Por exemplo, a metodologia KAOS [Dardenne et al., 1993] [Lamsweerde, 2000] oferece uma *framework* para elicitação de requisitos e gestão de sistemas. Em inglês KAOS significa *"Keep All Objectives Satisfied"*. O modelo de requisitos deste método é composto por quatro sub modelos: modelo de objectivos, modelo de objectos, modelo de agentes e um modelo de operação. Estes modelos são elaborados metodicamente usando uma abordagem orientada a objectivos [Lamsweerde, 2001]. Nesta metodologia realça-se a importância de representar e modelar explicitamente os objectivos organizacionais, e as suas relações com os componentes operacionais do sistema estão devidamente declaradas. KAOS está vocacionada para o processamento de requisitos [Dardenne et al., 1993] [Lamsweerde, 2000].

No KAOS, os objectivos podem ser atribuídos aos agentes através de ligações de responsabilidade AND ou OR. Isto permite a investigação de alternativas na fronteira entre o software e o seu ambiente. Uma ligação de responsabilidade entre um agente e um objectivo significa que o agente pode comprometer-se a realizar as suas operações, sob as condições (pré, post e trigger) que garantem o objectivo [Dardenne et al., 1993].

Esta metodologia tem sido aplicada em ambientes industriais nos mais variados domínios como as telecomunicações, imprensa, indústria alimentar e na indústria do aço [Ponsard, 2004]. Tipicamente, os documentos de requisitos vão de algumas dezenas a centenas de objectivos e são geridas pela ferramenta case do KAOS, o Objectiver[1].

O propósito é construir um sistema que satisfaça todas as necessidades dos *stakeholders*. As funcionais e as não funcionais. As não funcionais podem-se classificar da seguinte forma: o sistema deve ser seguro, barato, utilizável e eficiente. Deve ainda preservar o seu ambiente e respeitar com as leis em vigor. Os grafos desta metodologia

[1] http://www.objectiver.com

são grafos orientados permitindo que um objectivo possa aparecer em vários grafos para refinar objectivos de mais alto nível.

2.5.2 – V-graph

Outra abordagem orientada a objectivos é o V-graph. Trata-se de um modelo composto por três níveis de abstracção: *softgoals*, objectivos e tarefas. Este modelo permite a descrição de nós intencionais (objectivos e *softgoals*) e de nós operacionais (tarefas). Cada elemento no V-graph é dividido em tipo e tópico [Silva e Leite, 2005]. O tipo descreve uma função genérica, ou um requisito não funcional genérico. O tópico captura a informação contextual para o elemento.

Os diferentes níveis de abstracção do V-graph permitem uma separação das características de maneiras diferentes, enriquecendo os relacionamentos transversais entre os objectivos. Mais do que isso, o tipo e o tópico dos elementos permitem que tenhamos uma visão de dados e de funções no mesmo modelo, sem prejudicar a simplicidade visual das características [Silva e Leite, 2005].

2.6 – Sumário

Os actuais paradigmas de desenvolvimento começam a apresentar problemas quando aplicados ao desenvolvimento de software de grande porte. Entre eles, destaque para a capacidade para lidar com a mudança de requisitos, funcionais e não-funcionais. Nessa linha de pensamento, as metodologias orientadas a agentes apresentam-se como uma nova abordagem para construir sistemas complexos, que exige um esforço menor por parte dos engenheiros de software que as abordagens orientadas a objectivos.

No entanto, ainda não é possível usufruir as capacidades do paradigma de agentes, uma vez que ainda não existe um processo de desenvolvimento padronizado para construir software orientado a agentes. Isto faz com que seja difícil projectar e implementar sistemas multi-agentes.

Assim, o projecto Tropos propõe uma metodologia para desenvolver sistemas multi-agente que é centrada em requisitos e orientada a objectivos. O Tropos definiu estilos arquitecturais organizacionais que focam em processos de negócio, bem como em requisitos não-funcionais, e está vocacionada para aplicações multi-agentes. Esta metodologia apresenta algumas limitações na captura da informação necessária para projectar a arquitectura de SMA uma vez que usa o i* para representar os seus modelos arquitecturais.

Contrapondo a estas dificuldades, o UML para além de estar a ser estendido para suportar o desenvolvimento orientado a agentes, também vem sendo usado para representar a arquitectura de sistemas simples e complexos. Reconhecendo isso, o trabalho que já foi feito acerca do mapeamento dos conceitos do i* para os do metamodelo de Agência, forma a base desta dissertação. O objectivo é propor uma abordagem para transformar os estilos arquitecturais em i* para diagramas arquitecturais descritos de acordo com o perfil de Agência, através dos mecanismos oferecidos pelo MDE. Para além disso, com a consciência de que as heurísticas de mapeamento ainda não se encontram completas, pretende-se também melhorar o conjunto de heurísticas já existentes para guiar o processo de transformação.

3 – Desenvolvimento de Software Orientado a modelos

Este capítulo descreve a base das abordagens de desenvolvimento de software que são conhecidas como Orientadas a Modelos, nomeadamente a descrição do conceito de transformação no contexto de MDD. No final do capítulo é apresentado um breve estudo sobre algumas ferramentas de transformação de modelos.

3.1 – Introdução

O desenvolvimento de software é um processo complexo e muitas vezes de difícil execução, por se tratar de uma área multi-disciplinar da Informática. Desde a modelação e desenho à geração de código, gestão de projectos, testes, implementação em empresas, e gestão de alterações. Além disso, temos vindo a assistir à crescente complexidade de software, o que muitas vezes torna difícil a viabilidade de níveis elevados de qualidade, adaptabilidade face às mudanças, bem como a competitividade no custo de produção do software.

Existem algumas ferramentas que dão apoio ao Desenvolvimento Orientado a Modelos (*MDD – Model-Driven Development*) [Kleppe et al., 2007] para a plataforma Eclipse[2] [Kotonya e Sommerville, 1997] [Merks, 2004]. O MDA é uma abordagem ao desenho de software da OMG (do inglês, *Object Management Group*). O Desenvolvimento Orientado a Modelos é uma abordagem da engenharia do software que consiste na aplicação de transformações a modelos, de forma a elevar o nível de abstracção no qual as equipas de desenvolvimento criam e fazem evoluir o software. MDD impõe uma estrutura, e um vocabulário próprio, de modo a que estes artefactos primários (modelos) alcancem o seu principal propósito em cada passo do ciclo de vida do software. É neste aspecto que se encontra a principal diferença entre MDD e MDA. O MDA é mais restritivo. A linguagem de modelação do MDA é o UML enquanto o MDD não impõe qualquer restrição no que diz respeito a linguagem de modelação [OMG, 2001].

[2] http://www.eclipse.org/modeling/gmf/

Vários tipos de modelos podem ser criados dependendo da sua intencionalidade. Podem procurar um claro entendimento do problema, comunicação de vistas comuns do problema ou da sua solução, analisar aspectos chave da solução de uma forma formal ou informal, gerar detalhes de implementação de baixo nível a partir de um modelo de programação de alto nível entre outras [Brown, 2006]. Os modelos são altamente dependentes das características do domínio do problema ou da solução. Por exemplo, podem depender de relações estáticas entre os elementos do domínio, comportamentos dinâmicos de actores chave no sistema, aspectos operacionais do sistema a implementar, parâmetros de avaliação de desempenho entre outras [Brown, 2006]. As relações entre estes modelos oferecem uma rede de dependências que mantém o registo do processo pelo qual a solução foi criada e ajuda a perceber as implicações de mudanças em qualquer ponto do processo [Brown, 2006].

É o conjunto de modelos, modelação e de transformações de modelos que forma a base de um grupo de abordagens de desenvolvimento de software que são conhecidas como Orientadas a Modelos.

3.2 – PIM, PSM e geração de código

Depois de definida a arquitectura, a OMG define basicamente dois tipos de modelos. Os que levam em linha de conta as plataformas de hardware e de software sobre as quais se trabalha, e os modelos que não as consideram. Em MDA é possível pensar numa aplicação sem nos preocuparmos com a plataforma onde esta vai ser implementada. Assim surge o conceito de Modelos Independentes da Plataforma (do inglês, *Platform-Independent Model (PIM)*). Depois surgem os modelos que contêm informações acerca das tecnologias usadas na implementação. Estes, por exemplo, podem descrever a forma como um software pode ser desenvolvido em diferentes linguagens de programação com as particularidades que daí advêm.

Uma especificação completa em MDA consiste de um PIM e de vários Modelos Específicos da Plataforma (*Platform-Specific Models (PSM's)*) e conjuntos de definições de interfaces. O objectivo destes últimos modelos é descrever como o modelo base é implementado em diferentes plataformas.

O MDA começa com um modelo que se preocupa com a funcionalidade e o comportamento do sistema, independentemente das tecnologias sobre as quais o sistema vai ser implementado [Hailpern e Tarr, 2006]. Assim, o MDD faz a separação entre as funcionalidades e os detalhes de implementação. As ferramentas do MDA oferecem o

mapeamento dos PIM's para os PSM's através de técnicas de transformação. Este mapeamento surge à medida que novas tecnologias aparecem, ou quando as decisões de implementação mudam. É composto por um conjunto de regras que são usadas para modificar um modelo de forma a produzir outro modelo. Estas regras são usadas em transformações PIM-PIM, PIM-PSM, PSM,PSM e PSM-PIM.

A figura 3.1 procura representar graficamente as transições entre os vários estados do processo MDD, bem como as transformações envolvidas para a obtenção de código de forma automática.

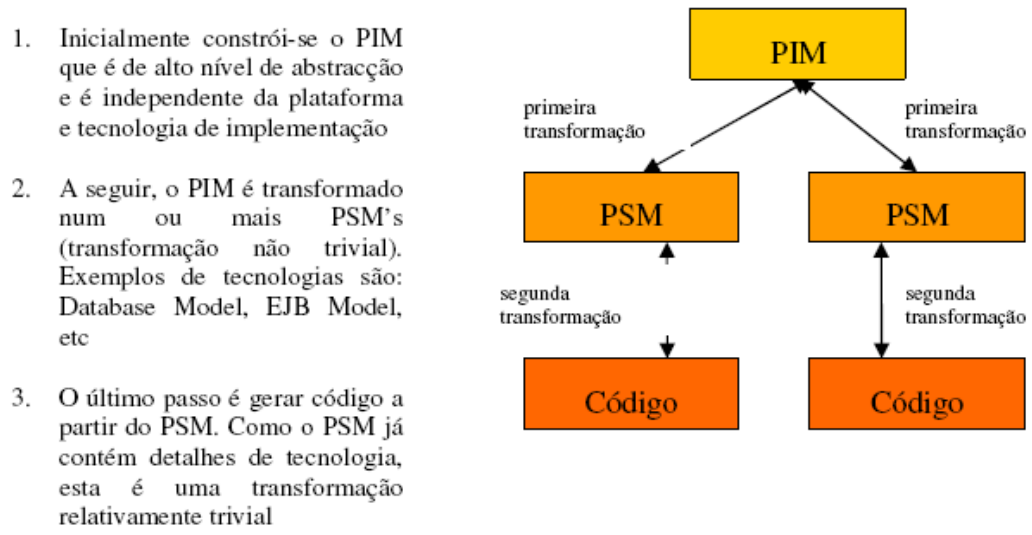

Figura 3.1 – PIM, PSM e Geração de código automática (figura retirada de [Santos, 2007])

Os vários níveis presentes na figura 3.1 representam níveis de abstracção. Pode-se dizer que ao nível dos modelos independentes da plataforma (PIM), será como programar num nível de abstracção superior, onde não são necessárias preocupações associadas às tecnologias a utilizar. Estas serão tratadas no nível abaixo, com a elaboração dos modelos dependentes das plataformas (PSM's).

3.3 – Transformação entre modelos

No contexto do MDD, a transformação de modelos pode ser classificada de acordo com duas características ortogonais: os formalismos dos modelos de entrada e de saída, e ao nível de abstracção. Olhando para a primeira característica temos transformações que são [Lara e Guerra, 2005]:

- Inter-Formalismos (Exógena) [Mens et al., 2004] – traduz o modelo para um formalismo diferente. Estas transformações têm que ver com bases de dados

ou migração de linguagens. Podem também ser utilizadas com finalidade de análise. Isto acontece quando o formalismo alvo possui métodos de análise que podem ser usados para estudar as propriedades do modelo original. Neste caso, a transformação deve preservar as características que se pretendem estudar.

- Intra-Formalismo (Endógena) [Mens et al., 2004] – aqui, quer o modelo original, quero o modelo alvo encontram-se no mesmo formalismo. Estas são transformações tipicamente usadas para optimizações, simplificações e *refactoring*.

Relativamente ao nível de abstracção existem transformações que podem ser:

- Horizontal – convertem o modelo noutro, no mesmo nível de abstracção. No contexto do MDA, implementam transformações PIM-PIM. As transformações para refactoring (intra-formalismo) ou migração (inter-formalismo) pertencem a esta categoria.
- Vertical – traduzem o modelo original num outro a um nível de abstracção diferente. Geração de código pode ser vista como uma transformação de um modelo de nível de abstracção mais alto para um mais baixo. No contexto do MDA, transformações PIM-PSM podem ser consideradas como verticais. No sentido contrário, a "*Reverse Engineering*" é uma transformação vertical de baixo para cima em termos de nível de abstracção.

A forma de expressão das transformações de modelos pode ser feita de várias maneiras. Pode-se mesmo dividir esta forma de expressão em três características principais: visual ou textual, imperativa ou declarativa e formal ou semi-formal. É possível a existência de metodologias híbridas em cada uma destas características.

Uma abordagem possível é a de "*graph transformation*" [Baresi e Heckel, 2002]. Esta encontra-se na categoria das linguagens visuais, declarativas e formais. Já a linguagem Java faz parte da categoria das linguagens textuais, imperativas e semi-formais.

Independentemente das categorias, existem algumas propriedades que são importantes em transformação de modelos. Por exemplo, terminação num período de tempo finito, confluência, cada modelo original gera apenas um modelo alvo, consistência sintáctica, o modelo alvo está de acordo como o metamodelo dado e

consistência semântica, propriedades semânticas, como o comportamento, são preservadas [Lara e Guerra, 2005].

3.4 – Metamodelação

O metamodelo define as restrições de um modelo. Pode ser usada como uma regra para os objectos ao nível do modelo, para os atributos dos objectos e a forma como estes se relacionam entre si. Os modelos específicos do domínio podem ser criados através da personalização do metamodelo [Bézivin et al., 2006].

Um metamodelo é um modelo de modelos, que introduz um novo nível de abstracção. Um modelo é apenas uma representação e, como tal, pode-se encontrar uma sequência virtualmente infinita de representações de representações, sem se sair do mesmo nível de abstracção. Ou seja, um modelo de um modelo, só pode ser considerado metamodelo, se for composto pelos conceitos subjacentes ao nível de modelação inferior. Um metamodelo descreve a possível estrutura dos modelos, define uma linguagem de modelação e as suas relações, assim como restrições e regras de modelação sem nunca definir uma sintaxe concreta dessa linguagem. Um modelo pode então ser visto como sendo uma instância possível de um determinado metamodelo. Os metamodelos definem portanto a sintaxe abstracta e semântica de uma determinada linguagem de modelação. Para definir um metamodelo, é então necessária uma linguagem de metamodelação, que por sua vez é descrita através de um metametamodelo. A figura 3.2 representa os três conceitos chave: Metametamodelos, metamodelos e modelos [Bézivin et al., 2006]. Assim, os metametamodelos (M3) definem a semântica dos metamodelos e de eles próprios. Os metamodelos (M2) estão conforme a especificação em M3 e definem a semântica dos modelos. Os modelos (M1) estão em conformidade com o especificado na camada M2 e representam o mundo real, os sistemas. Finalmente, ao nível M0 temos as instâncias dos modelos. A estas são atribuídos valores e atributos concretos conforme o estipulado ao nível M1.

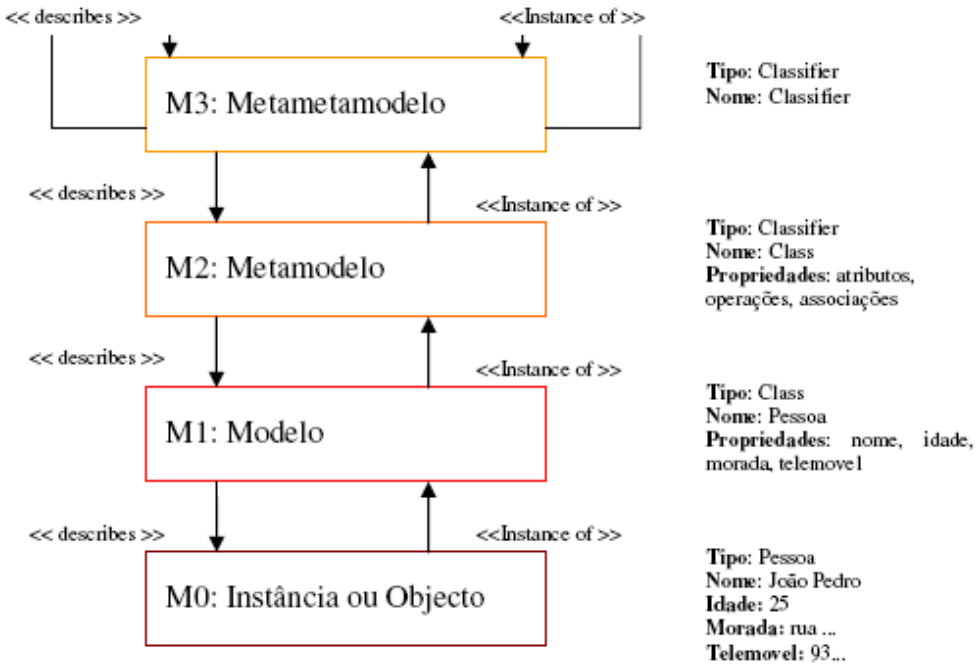

Figura 3.2 – Diagrama representativo dos 4 níveis de abstracção propostos pela OMG (figura retirada de [Santos, 2007])

Os sistemas aqui considerados são programas. Mas um programa também pode ser visto como um modelo que representa o mundo real. Por isso, podem ser transformados de modo a obter a concordância com o seu metamodelo específico. Os níveis representados na figura identificam os quatro níveis de abstracção propostos pela OMG, presentes no MDA.

Assim, o termo metamodelação é frequentemente utilizado para referir a actividade de criar modelos de processos de desenvolvimento de sistemas de informação. Metamodelação é a chave para a definição de linguagens específicas do domínio (DSL – *Domain-Specific Languages*). É um dos aspectos mais importantes quando se fala em desenvolvimento de software orientado a modelos. Para além disso, conhecimento acerca de metamodelação é necessário para a realização de outras tarefas em Engenharia de Software [Santos, 2007]. Entre elas, realce para a validação de modelos, transformações entre modelos, geração automática de código e integração de ferramentas.

No contexto do Desenvolvimento Orientado a Modelos, as linguagens específicas do domínio são definidas através de um metamodelo. A sintaxe concreta, ou seja, a forma concreta, gráfica ou textual, que se usa na construção de modelos, é conceptualmente irrelevante. Apenas tem de fazer a interpretação dos modelos de uma

forma não ambígua. Esta distinção entre sintaxe abstracta e concreta é muito importante, pois é o metamodelo, e não a sintaxe concreta, que forma a base para o processamento automático de modelos. Por outro lado, a sintaxe concreta é a interface oferecida para o utilizador para construir modelos. A qualidade de uma sintaxe concreta define a facilidade de leitura (do inglês, *readability*) de um modelo.

Na validação de modelos, estes são validados conforme as restrições impostas pelo metamodelo. Já as transformações entre modelos são definidas por mapeamento entre dois, ou mais, metamodelos. A geração de código automática é feita através da criação de *templates* que se referem ao metamodelo da DSL. A integração de ferramentas faz-se baseada no metamodelo. Assim, estas ferramentas podem ser adaptadas para o domínio do problema.

O EMF, do inglês *Eclipse Modeling Framework*, é um *plug-in* do Eclipse que permite a construção de metamodelos e as suas respectivas instanciações, modelos que se constroem em conformidade com o metamodelo [Merks, 2004]. Esta *framework* permite a criação de *plug-ins* através da capacidade que possui para geração automática de código.

3.5 – Ferramentas de Transformação

No desenvolvimento de software orientado a Modelos a transformação de modelos é uma actividade crítica. A esperada adopção generalizada da abordagem da OMG a este problema, a linguagem de transformação QVT [OMG, 2005], prevê-se que a experimentação em transformação de modelos aumente. No entanto, esta é apenas uma possível abordagem ao problema. Paralelamente existem alguns grupos de investigação a trabalhar nas suas próprias abordagens à transformação de modelos. É importante comparar e, posteriormente seleccionar, quais as linguagens mais adequadas para um problema em particular.

O evoluir destas abordagens tem o objectivo de desenvolver Linguagens Específicas de Domínio que são desenhadas para resolver as tarefas inerentes às transformações de modelos.

As linguagens de modelação, como o UML, são das mais usadas metodologias de abstracção em Engenharia de Software. O UML é uma linguagem de modelação de índole geral que oferece notação gráfica que é usada para criar um modelo abstracto de um sistema. A relevância do uso de UML para a modelação de sistemas é por demais conhecida, e as suas vantagens já foram brevemente discutidas anteriormente no

Capítulo 1. No entanto, o UML apresenta algumas fraquezas devido ao facto da sua sintaxe abstracta ser de carácter muito geral. Isto acontece porque o UML pretende modelar todos os modelos possíveis. Estes problemas tornam a geração de código mais difícil. É sabido que, na prática, é difícil ter linguagens de modelação de carácter geral com métodos de processamento eficientes [VMTS, 2008].

Estes problemas elevaram a importância das Linguagens Específicas de Domínio e, é através delas que se procura superar esses problemas. Estas linguagens tendem em suportar elevados níveis de abstracção, mais do que as linguagens de modelação de propósito geral. Para além disso, como as DSL's contêm apenas elementos específicos do domínio, permitem uma geração de código mais precisa. Deste modo requerem menos esforço e menos detalhes de baixo nível na especificação de um sistema [Vajk et tal., 2007].

Em MDA, o QVT (do inglês, *Queries, Views and Transformations*) [OMG, 2005] é um standard para transformação de modelos proposto pela OMG. Existem actualmente vários produtos que se apresentam como sendo compatíveis com o QVT. O QVT define o processo de transformar um modelo noutro. Várias ideias compõem a base desta proposta. Uma diz que os modelos inicial e final podem estar de acordo com metamodelos MOF arbitrários. Outra afirma que o programa que conduz a transformação é considerado como sendo um modelo e, por isso, deve estar de acordo com um metamodelo MOF. Isto faz com que a sintaxe abstracta do QVT tenha de estar de acordo com um metamodelo MOF.

O QVT define três DSL's: Relações (*Relations*), Core e Mapeamentos Operacionais (*Operational Mappings*). As relações e o core são linguagens declarativas que especificam as relações entre modelos e que fazem correspondência de padrões sobre um conjunto plano de variáveis respectivamente. O mapeamento operacional é uma linguagem imperativa que estende as duas anteriores. Estas três linguagens organizam-se numa arquitectura por camadas como o ilustrado na figura. As primeiras duas linguagens (Relações e Core) encontram-se em dois níveis diferentes de abstracção com mapeamentos definidos entre eles, enquanto os Mapeamentos Operacionais se apresentam como sendo transversais às duas primeiras linguagens declarativas.

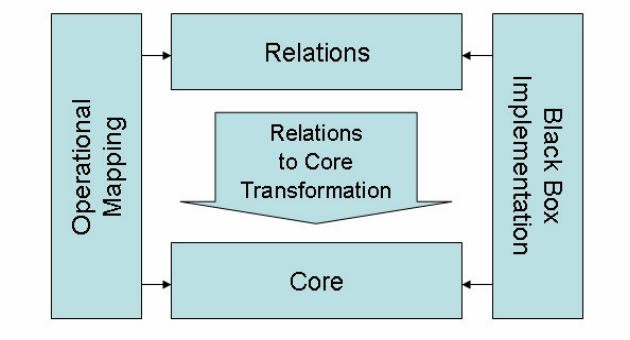

Figura 3.3 – Arquitectura QVT[3]

O mecanismo de implementação denominado "*Black Box*" existe para invocar transformações descritas em outras linguagens, tais como o XSLT ou XQuery. Este componente é particularmente útil na integração de bibliotecas externas ao QVT.

Até à data o QVT apenas se aplica a transformações entre modelos. Todas as transformações do tipo modelo para texto (por exemplo, XML) ou de texto para modelo ainda não são possíveis[4].

De entre as alternativas existentes em termos de linguagens de transformação, destaque para o ATL (*Atlas Transformation Language*), o VMTS (*Visual Modeling Transformation System*), e para o VIATRA2 (*Visual Automated Model Transformations*). Estas três ferramentas são das mais conhecidas no universo das ferramentas de transformação de modelos, em especial o ATL, e procuram criar uma sintaxe concreta para modelar sistemas de uma forma flexível e eficiente. Para esta dissertação será utilizada a linguagem ATL, por ser uma das mais utilizadas.

3.5.1 – Atlas Transformation Language (ATL)

ATL é uma linguagem de transformação de modelos usada na área da MDE. Tal como o VMTS, esta ferramenta permite produzir modelos a partir de um conjunto de modelos inicial. Trata-se de uma linguagem de transformação híbrida que permite construções declarativas e imperativas na definição das transformações [Jouault e Kurtev, 2006].

O padrão transformacional do ATL encontra-se representado na figura abaixo. Neste processo o modelo inicial Ma é transformado no modelo final Mb. Esta transformação é conduzida por uma definição da transformação a executar mma2mmb.atl. escrita na linguagem ATL. A definição da transformação é, ela própria,

[3] www.wikipedia.org
[4] *http://www.casetool.eu/qvt_en.html*

um modelo. Os modelos inicial e final, bem como a definição da transformação, estão de acordo com os seus metamodelos MMa e MMb e ATL respectivamente. Os metamodelos estão conforme o metametamodelo MOF como o descrito no Capítulo 3.

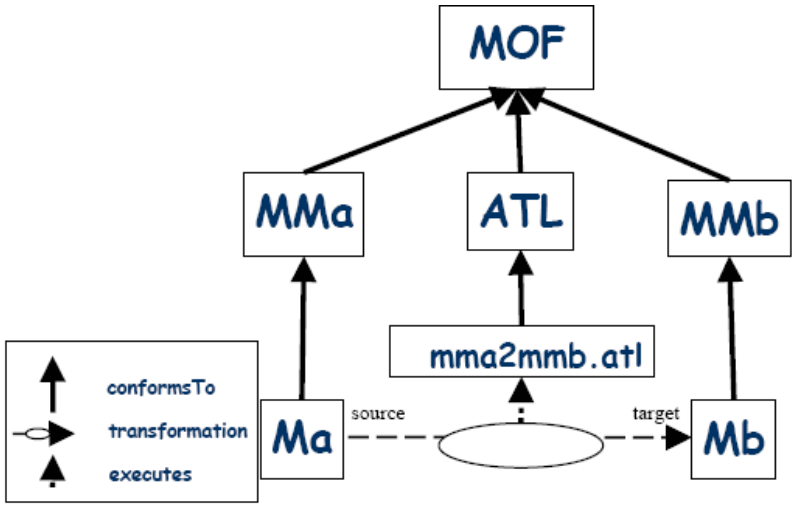

Figura 3.4 – Transformação de modelos em ATL [Jouault e Kurtev, 2006]

As transformações em ATL são unidireccionais e operam lendo os modelos de entrada e produzindo um modelo de saída. Durante a execução de uma transformação, o modelo inicial pode ser lido mas não alterado. O modelo de saída não pode ser lido nem é permitido navegar na sua estrutura. É possível implementar uma transformação bidireccional como sendo a composição de duas transformações. Uma em cada direcção [Jouault e Kurtev, 2006].

Os principais objectivos da *framework* ATL são permitir expressar especialização e composição de transformações e conseguir expressar transformações atómicas e compostas, sequências, pré e pós condições. Esta ferramenta procura também permitir definir transformações que tenham outras transformações como entrada [Bézivin et al., 2003].

O facto desta ferramenta oferecer dois tipos de metodologias de implementação (declarativa e imperativa) é a sua principal vantagem. Assim, pode-se usar uma abordagem declarativa para os problemas mais simples ou optar por uma imperativa quando a complexidade das transformações assim o exijam. A desvantagem desta ferramenta é não oferecer uma notação gráfica. É por isso uma ferramenta menos intuitiva do que, por exemplo, o VMTS. A criação visualização das instâncias dos metamodelos (modelos) é feita através do EMF. A interface oferecida pelo EMF é uma

esquematização em árvore, típica das aplicações xml, que se revela inadequada para construir modelos baseados em metamodelos não triviais. Relativamente ao tratamento de modelos baseados nas directrizes da *framework* i*, não existe nesta ferramenta, nem nas demais estudadas, uma característica específica que facilite esta visualização.

Ao nível da descrição dos metamodelos, o ATL permite uma abordagem visual bastante mais intuitiva, ainda assim com algumas limitações. Num estilo inspirado no UML, com as vantagens que reconhecidamente daí advêm, esta abordagem não permite associações bidireccionais. A bidireccionalidade apenas pode ser alcançada através de duas ligações unidireccionais. Para além disso, também não é possível definir cardinalidade nos dois extremos de uma associação, não sendo ainda claro visualmente a qual dos extremos da ligação se deve associar essa cardinalidade.

3.5.2 – Visual Modeling and Transformation System (VMTS)

De acordo com [VMTS, 2008], o VMTS é um ambiente de metamodelação por camadas. Esta ferramenta beneficia duma forte base matemática proveniente das linguagens formais e possui construções bem definidas para a especificação de regras de transformação, transformação de atributos e controlo de fluxo [Lengyel, 2006]. Para além disso, suporta diagramas UML. A técnica de transformação base usada para transformações com VMTS é a reescrita de grafos (do inglês, *graph rewriting*). É esse um dos pontos fortes desta ferramenta. Grafos oferecem uma representação expressiva e versátil dos dados.

Adoptando grafos como representação, é natural que a sua manipulação seja a base de computação utilizada. Estas manipulações podem ser representadas implicitamente, embebidas num programa que, entre outras coisas, constrói e modifica grafos. Em alternativa, manipulação de grafos pode ser representada explicitamente usando regras bem definidas de reescrita de grafos capazes de modificar o grafo original [Lengyel, 2006]. O uso explícito de regras de reescrita de grafos tem várias vantagens. Entre elas, destaque para o facto de permitir uma representação abstracta de alto nível de uma solução a um problema.

As transformações de modelos podem ser modeladas de forma visual. Assim, são facilmente modificadas e podem usufruir das vantagens que as técnicas de transformação de grafos bem definidas oferecem [Mezei et al., 2006]. O VMTS usa Processadores de Modelos Visuais (do inglês, *Visual Model Processors* – VMP's) para processar modelos com técnicas de transformação baseadas em reescrita de grafos. As

entradas do VMTS VMP são o modelo e metamodelo iniciais, o metamodelo de destino e o modelo de controlo de fluxo que define as transformações. Como resultado da transformação obtém-se o modelo de saída. O modelo inicial é uma instância do metamodelo inicial e o modelo de saída é uma instância do metamodelo de destino.

Esta ferramenta tem como ambiente gráfico o *VMTS Presentation Framework* (VPF), que permite visualizar e editar os modelos (grafos). Em VMTS a especificação de regras de transformação é feita através do VMTS *Rule Editor*. Trata-se de um plug-in que é integrado no VPF [Lengyel, 2006].

Em VMTS, as regras de transformação de grafos consistem em duas partes: LHS e RHS. Os grafos LHS e RHS baseiam-se na sintaxe dos diagramas de classes do UML. O primeiro é o metamodelo do modelo inicial, e o segundo é o metamodelo do modelo gerado. Assim, durante o processo da transformação, alguma parte do modelo inicial deve ser encontrada instanciando o LHS, e o modelo gerado será uma instância do RHS. As relações causais definidas entre os elementos do LHS e do RHS expressam a modificação, ou remoção, de um elemento do LHS e a criação de um elemento no RHS. Basicamente o motor de transformação procura no grafo inicial e, quando encontra uma ocorrência de LHS, substitui-a por pela respectiva parte da regra existente em RHS [Lengyel, 2006].

O componente principal do VMTS é o denominado "*Attributed Graph Architecture Supporting Inheritance*" (AGSI) que é a unidade de armazenamento de modelos. Esta usa um sistema de gestão de bases de dados relacionais para armazenar a informação dos modelos [Lengyel, 2006].

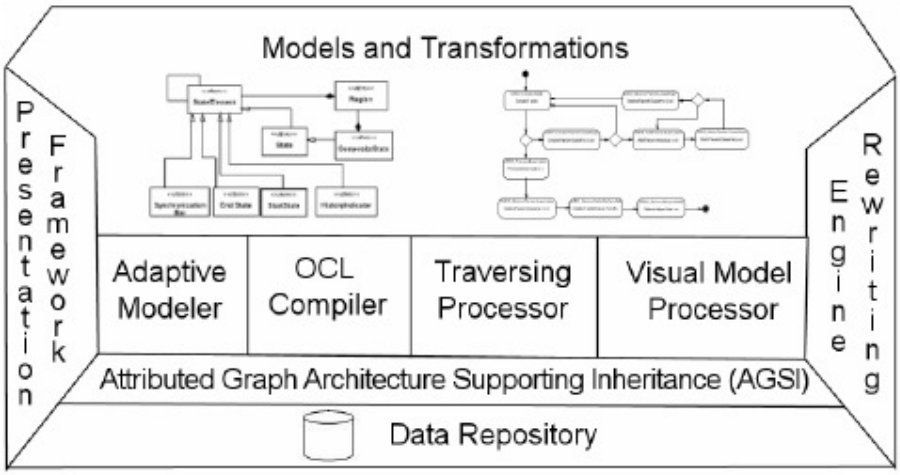

Figura 3.5 Estrutura do VMTS [Mészáros et al., 2007].

De acordo com [Madari, 2007], o VMTS tem uma arquitectura de *plug-in* que se direcciona para o desenvolvimento de *plug-ins* que permitem a comunicação com outras ferramentas (figura 3.5). O uso destes *plug-ins* permite relacionar os elementos dos modelos a uma representação gráfica de modo a tirar partido da interface intuitiva da ferramenta. Esta capacidade de usar *plug-ins* expande a abrangência do VMTS uma vez que permite a interacção desta ferramenta com outras. É possível a importação de metamodelos para VMTS, onde podem ser transformados, e permite a sua posterior exportação para a ferramenta de origem.

3.5.3 – VIATRA2 (Visual Automated Model Transformations)

De uma forma muito resumida pode-se dizer que o VIATRA2 é uma linguagem de transformação de grafos [Koch, 2006]. Mais concretamente, o VIATRA2 é uma *framework* de propósito geral para transformação de modelos, que procura suportar todo o ciclo de vida das transformações (especificação, desenho, execução, validação e manutenção) [Varró e Pataricza, 2004]. Esta definição apresenta-se actualmente como a melhor descrição do principal objectivo desta ferramenta.

Segundo o apresentado em [Varró e Pataricza, 2004], o VIATRA2 usa a abordagem de metamodelação VPM (do inglês, *Visual Precise and Multilevel*) em vez da habitual MOF para descrever linguagens de modelação e modelos. Isto acontece principalmente porque com VPM é possível modelar um número arbitrários de "metaníveis" no espaço do modelo. Com MOF tínhamos uma arquitectura de quatro níveis fixos. Como consequência directa desta flexibilidade, temos que é possível integrar modelos vindos dos mais variados domínios e/ou espaços tecnológicos.

No que diz respeito à especificação concreta das transformações, esta ferramenta combina um paradigma visual baseado em padrões com um paradigma formal de alto nível. *Queries* feitas sobre os modelos são intuitivamente capturadas pelos padrões de grafos. Estes definem restrições e condições nos modelos. Manipulações elementares de modelos podem ser definidas à base de regras de transformação de grafos enquanto as regras das máquinas de estados abstractas (do inglês, Abstract State Machines - ASM) podem ser usadas para a descrição de estruturas de controlo. Assim, transformações complexas podem ser conseguidas à base de construções baseadas ASM. Deste modo é possível combinar transformação de grafos com máquinas de estados abstractas num único paradigma [Varró e Pataricza, 2004].

A linguagem que foi criada para implementar estes conceitos é a VTCL (do inglês, *Viatra Textual Command Language*). Esta linguagem é primeiramente textual mas está a ser estendida por um editor gráfico que irá suportar as definições gráficas das transformações de modelos. A figura 3.8 ilustra esta abordagem.

Uma característica única do VIATRA2 é o suporte para transformações genéricas e meta-transformações [Varró e Pataricza, 2004] que permitem munir os parâmetros com tipo e manipular transformações como modelos normais respectivamente.

O processo de execução das transformações pode-se resumir a duas fases. Numa primeira fase as transformações são executadas dentro da *framework* através do interpretador do VIATRA2. Este interpretador usa técnicas de satisfação de restrições para conseguir correspondências de padrões. Numa segunda fase, processo decorre de uma forma similar ao VMTS. Há um grafo inicial de acordo com um determinado metamodelo, e temos um grafo de saída que deve ser gerado de acordo com um outro metamodelo. Pelo meio existem especificações de regras de transformação que permitem a passagem do modelo de entrada para o modelo de saída.

As transformações em VIATRA2 são unidireccionais mas já estão a ser desenvolvidos esforços no sentido de se conseguir uma abordagem bidireccional e incremental à transformação de modelos.

3.6 – Metamodelos: i* e Agência

Nesta subsecção apresentamos os metamodelos a serem utilizados nesta dissertação, nomeadamente o metamodelo do i* e o metamodelo de agência.

3.6.1 – Metamodelo do i*

Existe uma versão do metamodelo i* datada de 1995 [Yu, 1995]. No entanto outra foi especificada em [Alencar et al., 2008] usando o metamodelo MOF (do inglês, *Meta-Object Facility*) da OMG. A ideia foi incorporar novas abstracções de modo a esconder informação detalhada e produzir modelos mais simples.

De acordo com este metamodelo, os modelos i* são compostos de pelo menos um nó, que pode ser um Dependum ou um *DependableNode*. O *DependableNode* é especializado num Actor e num *InternalElement*. Significa isto que um Actor (ou um *InternalElement*) pode depender de um *InternalElement* que se encontre dentro de outro Actor, que pode ser ele próprio. Um *InternalElement* pode ser especializado em

Alternative ou em *IntencionalElement*. Este pode ser relacionado com *IntentionalElements* através de uma ligação *ContributionLink*, *MeansEndLink* ou *TaskDecompositionLink* [Alencar et al., 2008].

De notar que as novas construções da linguagem i*, propostas em [Alencar et al., 2008], são o *InternalElement*, a *Alternative* e dois novos atributos na *TaskDecompositionLink*. Um *InternalElement* é um qualquer elemento que dentro da fronteira de um Actor ou *Alternative*. Uma *Alternative* é uma metaclasse que foi adicionada para permitir a supressão de um sub grafo composto de, pelo menos, um *InternalElement*. Está também encarregada de *operacionalizar* um *InternalElement* através de uma relação *MeansEndLink*. Um sub elemento relacionado com uma tarefa, através de uma relação *TaskDecompositionLink*, pode ser ordenado por um atributo opcional de prioridade, o *priority*, que define a ordem de execução. Elementos com a mesma prioridade podem ser executados em paralelo. Existe um atributo que se torna redundante nestes casos, que é o atributo *parallel*. Este existe para indicar paralelismo explicitamente. O metamodelo i* encontra-se ilustrado na figura 3.6.

As classes "*Agent*", "*Role*" e "*Position*" são especializações do conceito de "Actor" que por sua vez é uma especialização das classes que representam o conceito de nó. Para esta dissertação considera-se apenas o conceito de *"Role"*, ignorando os conceitos de *"Agent"* e *"Position"* que não possuem nenhuma correspondência a nível do Diagrama Arquitectural. Um Actor participa numa, ou em mais do que uma "*IStarRelationship*", podendo exercer o papel de "*Depender*", ou "*Dependee*", no acesso a um dependum. Este último pode ser de quatro tipos (classe de enumeração "*DependumKind*"): *goal*, *softgoal*, tarefa ou recurso. Em última análise, os actores e os seus papéis nas relações entre si, dentro de um diagrama de Dependência Estratégica, são representados como "*Node*" e "*IStarRelationship*", que não são mais que os componentes que compõem um "*IStarModel*".

Deste metamodelo importa retirar apenas os conceitos envolvidos no diagrama de dependência estratégica. Serão esses conceitos que irão fazer parte do metamodelo de origem da nossa abordagem. Uma versão deste metamodelo será elaborada, contendo apenas as abstracções que vão ser alvo desta dissertação.

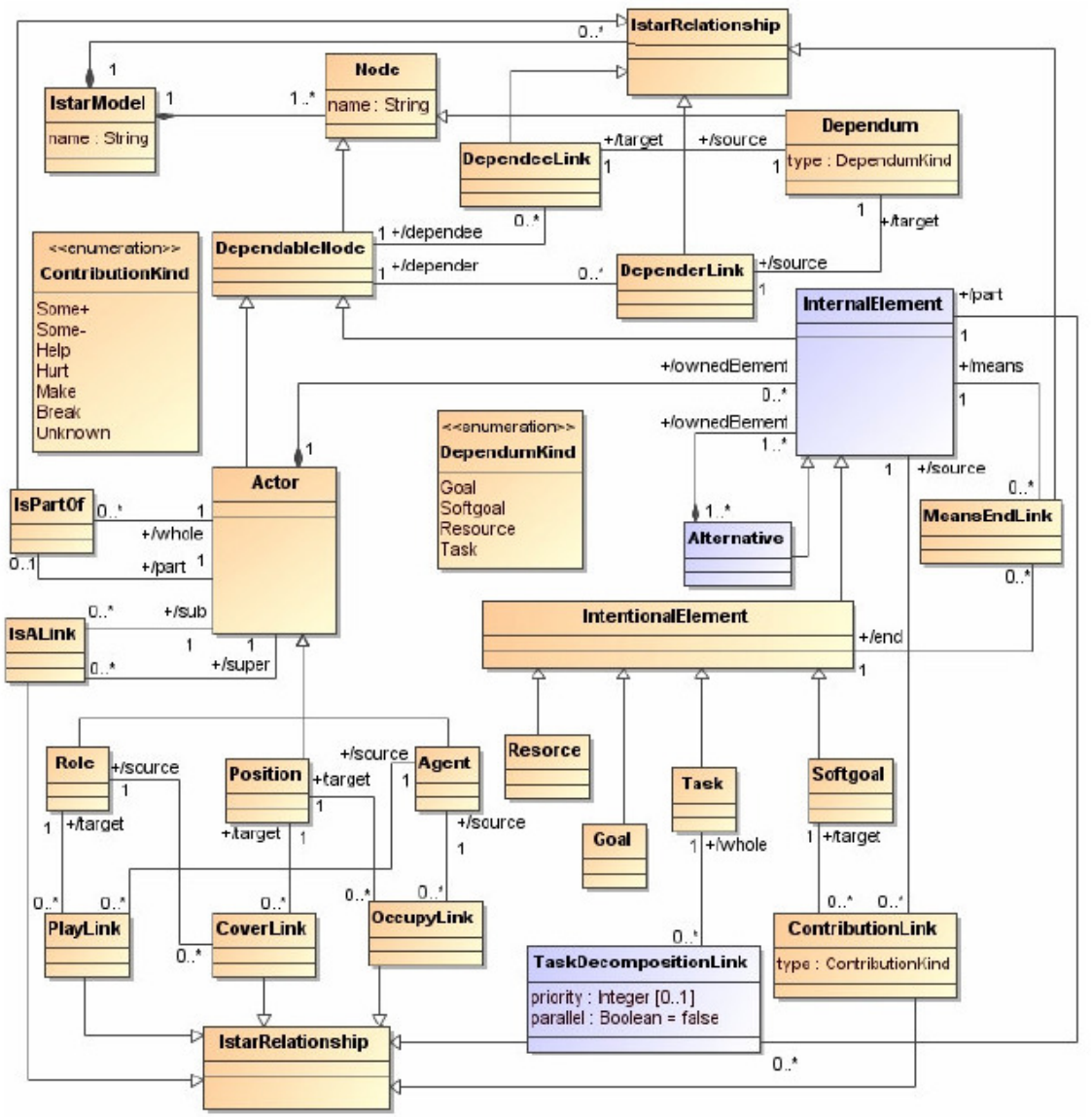

Figura 3.6 – Metamodelo i* [Alencar et al., 2008]

3.6.2 – Metamodelo de agência

Como a dissertação propõe transformações de modelos arquitecturais em i* para modelos arquitecturais descritos de acordo com o perfil de Agência, nesta sessão será explicada apenas a parte deste metamodelo que se refere ao diagrama arquitectural apresentado na secção 2.4.

Um SMA é concebido como uma organização composta por agentes. O conceito de agente estende a meta classe *Class* do pacote *StructuredClasses*, que por sua vez estende a meta classe *Class*, do pacote *Kernel*, com capacidade de ter uma estrutura interna e portas. É o AgentRole que faz com que um agente pertença a uma organização e que tenha direitos dentro da mesma.

Uma *ComplexAction* determina os passos que compõem um plano. Um *MacroPlan* é um plano definido por um *AgentRole* e é um plano parcial e como tal composto por *ComplexActions*. Os *AgentRoles* precisam trocar sinais através de um *AgentConnector* de forma a alcançarem o acordo contratual de fornecimento de serviço entre eles. Uma *OrganizationalPort* determina um ponto de interacção distinto entre o *AgentRole* e o seu ambiente. Um *AgentConnectorEnd* é ponto final de um *AgentConnector*, que agrega o *AgentConnector* à *OrganizationalPort*. Na figura 3.7 encontra-se ilustrada a parte do metamodelo de agência referente às abstracções contidas no diagrama arquitectural.

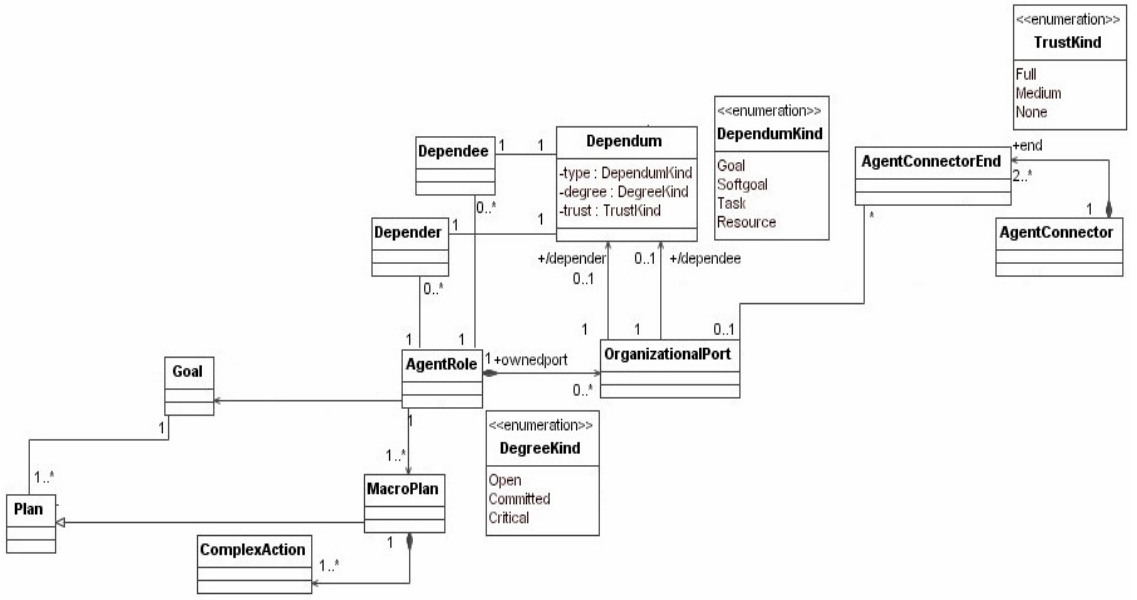

Figura 3.7 – Fragmento do metamodelo de agência – Metaclasses envolvidas no Diagrama Arquitectural

Alguns conceitos presentes são similares aos conceitos da *framework* i* já definidos no Capítulo 2. Um dependum define um acordo entre dois *AgentRoles*, que tomam o papel de depender e dependee. Assim, o *AgentRole* responsável por oferecer o serviço possui um *OrganizationalPort*, desempenha o papel de dependee e relaciona-se com o dependum através de uma relação dependee. O *AgentRole* que precisa do serviço tem um *OrganizationalPort*, desempenha o papel de *depender* e relaciona-se com o dependum através de uma relação depender. A meta classe *Dependum* possui a propriedade *trust* que mostra que o depender tem de confiar no dependee para delegação de actividades. A propriedade *trust* pode ser de três tipos: total (*full*), média (*medium*) ou nenhuma (*none*), como o definido na classe de enumeração *TrustKind*. As dependências entre *AgentRoles* possuem três diferentes graus, determinados pela classe de enumeração *DegreeKind*: aberta (*open*), o *depender* é pouco afectado pela falha do

dependum; *committed*, o *depender* é afectado de forma significativa pela falha do dependum, e crítica (*critical*), o *depender* é seriamente afectado pela falha do dependum [Yu, 1995].

É preciso notar que uma relação num modelo i* pode ter um mapeamento para qualquer um dos seis diagramas do perfil de agência apresentado anteriormente neste capítulo. Nesta dissertação apenas se pretende melhorar o conjunto de heurísticas que guiam o mapeamento do diagrama de dependência estratégica do i* para o diagrama arquitectural do perfil de agência em UML.

3.7 – Sumário

O MDA é uma *framework* de desenvolvimento definido pela OMG onde o conceito chave é o de modelo. Os modelos que formam a base desta *framework* são: os PIM, que são modelos a um nível mais alto de abstracção independentes das tecnologias de implementação; os PSM, que são modelos cujo intuito é especificar o sistema em termos de construções específicas de uma determinada tecnologia de implementação; e o código, que é uma descrição do sistema em código fonte. A grande vantagem é que as transformações desde o nível mais alto de abstracção até ao código são feitas através de transformações de modelos automatizadas utilizando ferramentas, facilitando a rastreabilidade no desenvolvimento do sistema.

O desenvolvimento orientado a modelos é uma tecnologia emergente. As ferramentas ainda não estão desenvolvidas ao ponto de conseguir uma geração de código tão eficiente que não exija uma intervenção humana.

Existem alguns benefícios importantes que fazem da tecnologia baseada em MDA uma boa escolha no desenvolvimento de software. Com a geração automática de código consegue-se poupar muito esforço às equipas de desenvolvimento e daí advêm ganhos de produtividade. O modelo PIM é independente da plataforma e permite obter um PSM oferecendo portabilidade para diferentes plataformas. Os modelos podem fazer parte da documentação de um sistema. As alterações nos modelos podem ser simples, a geração de código é automática e a documentação à base de modelos poderá ser útil na manutenção do sistema, para além de permitir uma maior rastreabilidade no desenvolvimento do sistema. Além disso, os modelos podem ser reutilizados noutros sistemas.

Sendo a transformação de modelos arquitecturais descritos em i* para modelos arquitecturais de Agência a intenção desta dissertação, pretende-se alcançar esse

objectivo aproveitando as facilidades oferecidas pelo MDA para automatizar essas transformações.

Ao nível das ferramentas nota-se ainda alguma instabilidade. É actualmente um grande problema da área do MDD. O suporte a nível de ferramentas ainda não é o mais adequado. Muitas das ferramentas propostas não funcionam correctamente e demonstram alguma falta de capacidade de exprimir todas as potencialidades da abordagem orientada a modelos. Por exemplo, o VIATRA2 apresenta problemas ao nível da instalação.

Após feito o estudo das ferramentas disponíveis, o ATL foi a escolhida para realizar as transformações entre modelos. Esta escolha foi influenciada pela estabilidade, funcionalidade e simplicidade de uso desta ferramenta Trata-se de uma ferramenta mais madura e com mais e melhor documentação que as demais estudadas. O VIATRA2 apresentou problemas logo ao nível da instalação enquanto o ATL revelou-se uma ferramenta mais intuitiva que o VMTS. Tendo em consideração a subjectividade desta avaliação, não quer isto dizer que a ferramenta escolhida seja melhor ou pior que as demais, mas demonstrou ser a mais adequada para o trabalho apresentado nesta dissertação.

Neste capítulo foram ainda apresentados os metamodelos de origem (*framework* i*) e de destino (diagrama arquitectural do perfil de Agência) das transformações a que me proponho realizar nesta dissertação. Estes dois modelos são usados para modelar a arquitectura de sistemas multi-agentes. Esta dissertação procura superar as actuais limitações existentes na modelação da arquitectura de SMA, transformando modelos arquitecturais descritos em i* (secção 3.6.1), para modelos arquitecturais descritos no perfil de Agência (secção 3.6.2).

Em particular, o diagrama arquitectural de Agência é capaz de representar as construções arquitecturais de um SMA, como por exemplo portas, conectores e interfaces. Para além disso, outras construções, como papeis (roles), confiança e grau de dependência também são modeláveis [Silva C., 2007].

Apresentados que estão os metamodelos, o capítulo seguinte apresenta uma análise das heurísticas existentes, novas heurísticas de mapeamento e as regras de transformação de modelos arquitecturais de i* para modelos arquitecturais de Agência.

4 – Transformando arquitecturas organizacionais em i* para arquitecturas baseadas em Agentes

Este capítulo descreve a contribuição principal desta dissertação: o processo de transformar modelos arquitecturais organizacionais descritos em i* para modelos arquitecturais definidos segundo o metamodelo de Agencia.

4.1 – Considerações iniciais

Para se conseguir um bom mapeamento entre duas linguagens, é necessário primeiro definir a sintaxe e semântica das mesmas. Um bom mapeamento entre duas linguagens só pode ser conseguido se estas se encontrarem clara e correctamente especificadas. Foi esse o objectivo do capítulo anterior, onde o metamodelo da *framework* i* e de parte do perfil de Agência em UML foram dissecados. De facto, apenas as metaclasses e as relações do metamodelo de Agência envolvidas na descrição do diagrama arquitectural foram analisadas porque esta dissertação propõe-se a apresentar um processo automatizado de realizar transformações de modelos SD do i* para modelos arquitecturais de Agência.

A razão de utilizarmos estes dois modelos particulares prende-se com o factor simplicidade. Tornar-se-à muito complexo trabalhar sobre a totalidade do metamodelo i* e de agência. Se é possível dividir os metamodelos em secções mais pequenas, é possível conseguir um processo de mapeamento com qualidade, que no futuro pode ser aplicado ao que resta dos modelos em causa.

Neste sentido, vão ser apresentados os metamodelos particulares que servirão de base para a contribuição desta tese, implementados em Ecore [Merks, 2004] , um plug-in do Eclipse bastante utilizado para metamodelação. O uso de Ecore prende-se com o facto de se tratar de um metamodelo implementado pela ferramenta Eclipse que vai ser utilizada para a realização das transformações. Neste capítulo, a versão Ecore dos metamodelos apresenta apenas as metaclasses e as relações que participam no diagrama de dependência estratégica e no diagrama arquitectural de Agência. Estes metamodelos

definem um conjunto de possíveis instanciações, que produzem modelos sintacticamente correctos em alguma linguagem de modelação.

De forma a tornar mais completo o processo de transformar arquitecturas organizacionais em i* para arquitecturas baseadas em agentes, procedemos também à elaboração de novas heurísticas de mapeamento, ao nível dos metamodelos. O objectivo foi tornar o conjunto de heurísticas completo e capaz para o sucesso da nossa abordagem.

4.2 – Heurísticas de mapeamento i*

Para guiar o processo de transformação de modelos arquitecturais descritos em i* para outros descritos de acordo com o metamodelo de Agência, torna-se necessário definir um conjunto de heurísticas. Existem já alguns trabalhos feitos nesse sentido [Silva C., 2007] [Silva C. et al., 2006a] no entanto, as heurísticas já definidas não são ainda exaustivas e estão ainda sujeitas a melhoramentos, como discutido de seguida.

4.2.1 – Heurísticas já existentes

Apresenta-se de seguida algumas heurísticas já existentes para a condução deste processo de transformação. Como já foi referido na secção 2.4, em [Silva C. et al., 2006a] definiram-se algumas heurísticas. Em [Silva C., 2007] essas heurísticas foram melhoradas levando em linha de conta os conceitos arquitecturais dos SMA (tabela 4.1):

Tabela 4.1 – Levantamento das heurísticas definidas em [Silva C., 2007]

#	*Framework* i*	UML	Comentários
#1	Papel (role)	Classe «AgentRole»	Um papel em i* representa-se como um papel desempenhado por um Agente em UML. Diagrama arquitectural
#2	Agent	Classe «Agent»	O conceito de agente é idêntico nas duas abordagens. Diagrama intencional
#3	Agent -> Role "desempenha"	Classe associativa «Intention»	Entre o «Agent» e o «AgentRole». Diagrama intencional.
#4	Tarefa	Classe «MacroPlan»	Representa os meios pelos quais se alcança um *goal*. Diagrama intencional e arquitectural.
#5	Sub-tarefa	Classe «ComplexAction»	Representa os passos que compõem um «MacroPlan». Diagrama arquitectural.
#6	Softgoal	Classe «Softgoal»	Apenas para os *Softgoals* não decompostos. Diagrama Racional

#7	Contribuição de uma tarefa para um *softgoal*	Classe associativa «Contribution»	Entre o correspondente «MacroPlan» e «Softgoal». Diagrama racional
#8	Softgoal (como parte de uma tarefa)	Classe «Goal»	Cria uma «ComplexAction» responsável por gerar esse *goal* no «MacroPlan» correspondente à tarefa.
#9	Dependum	Interface «Dependum»	Um *dependum* pode ser de quatro tipos: *goals*, *softgoals*, tarefas e recursos. Diagrama arquitectural.
#10	Depender	Dependência «Depender»	A posição de depender em i* torna-se uma dependência com o mesmo nome. Diagrama arquitectural.
#11	Dependee	Realização «Dependee»	A posição *dependee* em i* representa-se como uma realização de interface com o mesmo nome. Diagrama arquitectural.
#12	Depender – dependum – dependee	Associação «Conector»	Portas são adicionadas aos agentes para proporcionarem um *link* para o conector. Diagrama arquitectural.
#13	Recurso	Classe «Resource»	Representa um recurso ambiental do qual o agente precisa para alcançar objectivos. Diagrama ambiental.
#14	Goal/Softgoal	Classe «Goal»	Aplica-se apenas a *goals* / *softgoals* não decompostos. Representa os objectivos do sistema. Diagrama arquitectural e intencional.
#15	Softgoal	Classe «SoftGoal»	Aplica-se apenas a *softgoals* não decompostos. Diagrama Racional.
#16	Belief	«Plan» ou «AgentAction»	Uma *belief* é uma condição para realizar uma tarefa. Representa o conhecimento que um agente possui acerca de si e do seu ambiente. Diagrama arquitectural.
#17	Organização	SMA	O sistema ao qual um agente pertence. Uma organização é composta de AgentRoles e outras organizações.

Para além das heurísticas descritas na tabela 4.1, outra heurística definida em [Silva C., 2007]: Uma dependência num modelo i* é (i) aberta se tiver o símbolo "o"; (ii) *committed* se não tiver símbolo; (iii) crítica se tiver o símbolo "x". Essa informação pode ser capturada na propriedade grau (*degree*) da classe «Dependum».

Estas são as heurísticas que existem actualmente para nos ajudar no processo de transformar modelos arquitecturais descritos em i* para modelos arquitecturais descritos

em UML. Servem estas heurísticas para ilustrar o trabalho que tem vindo a ser feito nesta área, e a abrangência que se pretende que estas heurísticas tenham no domínio deste problema. É preciso notar que uma relação num modelo i* pode ter um mapeamento para qualquer um dos seis diagramas do perfil de agência (Arquitectural, Comunicação, Ambiental, Intencional, Racional e Acção), do qual apenas vamos tratar o diagrama arquitectural, apresentado anteriormente neste capítulo.

Por isso, nesta dissertação, apenas se considera o conjunto de heurísticas que guiam o mapeamento do diagrama de dependência estratégica do i* para o diagrama arquitectural do metamodelo de agência.

Como veremos na secção 4.3, os metamodelos dos diagramas SD e Arquitectural, possuem classes enumeradas. No entanto, actualmente, as heurísticas não consideram os mapeamentos dos seus valores. Trata-se de um mapeamento simples e directo que não se encontrava suficientemente definido.

No diagrama arquitectural é introduzido o conceito de confiança (do inglês, *trust*) nas relações. Esta introdução não se encontra devidamente reproduzida ao nível das heurísticas. O conceito de confiança pode ser visto como dependendo directamente do atributo *degree* das relações do i*. Na secção seguinte é dada uma sugestão para esta insuficiência.

Outro conceito não considerado no conjunto actual de heurísticas é o de herança. Como vimos anteriormente no exemplo do *Meeting Scheduler*, usado para introduzir o i*, podem haver relações de herança (IsA) entre actores. Naquele exemplo, um actor *Important Participant* era considerado como uma especialização de *Meeting Participant*, com a particularidade da sua presença ser essencial a qualquer reunião em que participasse. Actualmente essa possibilidade não era considerada a nível de heurísticas de mapeamento. Como vamos ver mais à frente, este conceito de herança obrigou igualmente a uma pequena alteração ao nível do metamodelo, dado que o metamodelo também não considerava esta particularidade.

Tomando consciência das limitações ainda existentes ao nível dos metamodelos e das heurísticas existentes actualmente, importa agora procurar superá-las. Nesta secção identificámos concretamente as falhas que existem, vamos agora passar à nossa proposta para superar essas limitações.

4.3 – Novas heurísticas de mapeamento

Depois de discutir os metamodelos particulares que vão ser objecto de estudo nesta dissertação (secção 3.6) e as respectivas heurísticas de mapeamento, apresentamos agora um estudo com o objectivo de enriquecer aquele conjunto de heurísticas. Um dos propósitos desta dissertação é procurar torná-las claras e completas e procurar superar as lacunas previamente identificadas na secção anterior.

Nesta fase, analisámos as propriedades essenciais das transformações que pretendemos automatizar, apresentamos (figuras 4.4, 4.2 e 4.3) algumas novas heurísticas de mapeamento que se baseiam nas ideias apresentadas na secção anterior e visam completar as já existentes, anteriormente apresentadas. Estas novas heurísticas servem para introduzir regras para modelar características dos metamodelos, SD e arquitectural, que não eram levadas em linha de conta em [Silva C., 2007]. Entre essas características encontramos regras que modelam a relação que existe entre os conceitos de *degree* e *trust* nesses diagramas, figura 4.1. Já a figura 4.2 trata de uma heurística que mapeia os diferentes tipos de relações existentes em i* para os valores de *DependumKind* do metamodelo de Agência. Por último, a figura 4.3 define uma heurística para a noção de herança. Este conceito motivou, inclusive, uma alteração ao nível do metamodelo de Agência que não incluía esta abstracção de herança na sua definição. Trata-se da incorporação de um novo conceito ao nível das heurísticas conhecidas até à data.

<table>
<tr><th colspan="3">Degree – Trust</th></tr>
<tr><th>Diagrama SD</th><th>Diagrama Arquitectural</th><th>Descrição</th></tr>
<tr><td>Dependência Crítica</td><td>Trust = Full
Degree = Critical</td><td>Valor do atributo trust e degree da classe Dependum. Diagrama arquitectural</td></tr>
<tr><td>Dependência Aberta</td><td>Trust = None
Degree = Open</td><td>Valor do atributo trust e degree da classe Dependum. Diagrama arquitectural</td></tr>
<tr><td>Dependência Normal</td><td>Trust = Medium
Degree = Committed</td><td>Valor do atributo trust e degree da classe Dependum. Diagrama arquitectural</td></tr>
<tr><td colspan="3">Se uma dependência no diagrama de Dependência Estratégica, independentemente do seu tipo, for marcada como “Critical”, o atributo degree na classe Dependum terá a mesma designação e o atributo trust na mesma classe é definido como Full.
Se essa dependência estiver representada como “Open”, o atributo degree na classe</td></tr>
</table>

Dependum terá a mesma designação e o atributo *trust* da mesma classe é decretado como *None*. Caso contrário é fixado o valor de *trust* como sendo *Medium* e o valor de *degree* é *committed*.

Mesmo assim, esta heurística pode ser vista como um padrão geral aplicável à maioria dos sistemas. Pode-se dar o caso em que todas as dependências entre actores, que não só as críticas, exijam um grau de confiança *full*.

Figura 4.1 – Heurística degree-trust

Tipo de Dependência – DependumKind		
Diagrama SD	Diagrama Arquitectural	Descrição
Dependência de Goal	Valor "goal" na classe de enumeração DependumKind	Uma dependência de *"goal"* implica que o valor *goal* esteja contido dentro dos valores da classe de enumeração *DependumKind*. Diagrama arquitectural
Dependência Softgoal	Valor "softgoal" na classe de enumeração DependumKind	Uma dependência de *"softgoal"* implica que o valor *softgoal* esteja contido dentro dos valores da classe de enumeração *DependumKind*. Diagrama arquitectural
Dependência de Tarefa	Valor "task" na classe de enumeração DependumKind	Uma dependência de tarefa implica que o valor *task* esteja contido dentro dos valores da classe de enumeração *DependumKind*. Diagrama arquitectural
Dependência Recurso	Valor "resource" na classe de enumeração DependumKind	Uma dependência de *"resource"* implica que o valor *resource* esteja contido dentro dos valores da classe de enumeração *DependumKind*. Diagrama arquitectural

No diagrama de dependência estratégica existem quatro tipos de dependência: dependência de *goal*, de *softgoal*, de recurso e de tarefa. Estas dependências têm o nome do tipo de dependum (*DependumKind*) que as compõem. Assim, pode-se definir um mapeamento directo entre os tipos de dependum existentes em i* para uma classe de enumeração DependumKind no diagrama arquitectural. Como já foi visto no Capítulo 2, o tipo de uma dependência é determinado pelo tipo do Dependum envolvido nessa relação. Todos esses tipos têm de ser tidos em consideração no diagrama arquitectural do perfil de agência. Ao nível do metamodelo, estes tipos são tratados ao nível da classe de enumeração *DependumKind*. Assim, se uma relação entre dois Actores é por um *goal*, a

dependência é dita de *goal* (*Goal Dependency*), e a classe *DependumKind* precisa conter esse valor como um dos seus enumerados. O mesmo raciocínio se aplica aos restantes três tipos de dependência. Desta forma, por cada tipo de dependência existe um valor enumerado correspondente na classe de enumeração *DependumKind*.

Figura 4.2 – Heurística tipo de dependência

Herança		
Diagrama SD	Diagrama Arquitectural	Descrição
Relação IsA	Relação ISA	Mapeamento directo da generalização. Diagrama arquitectural
A noção de herança é idêntica em ambos os diagramas. Se, no modelo SD, um Actor é uma especialização de outro, os AgentRoles correspondentes terão a mesma relação de herança entre si.		

Figura 4.3 – Heurística para herança

4.4 – Metamodelo Ecore do diagrama de Dependência Estratégica

Como o referido no Capítulo 2, o Tropos é uma metodologia de desenvolvimento de software baseada nos conceitos chave das abordagens orientadas a agentes. Em particular, no metamodelo do diagrama de Dependência Estratégica podemos encontrar metaclasses e relações que modelam conceitos como dependência, *goal*, *softgoal*, recurso, tarefa ou actor. Este último pode ser visto como o conceito central neste diagrama já que tudo gira em torno dele. O diagrama de Dependência Estratégica já foi apresentado no Capítulo 2.

Do modelo SD fazem parte um conjunto de actores e dependências entre eles, com ênfase nas suas intenções e responsabilidades. Essas dependências podem ser por um recurso, uma tarefa, um *goal* ou um *softgoal*. Em última análise pode ser visto como uma rede de dependências entre nós (actores).

A figura 4.4 ilustra o metamodelo do diagrama de Dependência Estratégica definido em Ecore. Este modelo apresenta algumas diferenças relativamente ao seu homólogo em MOF. Para efeitos de identificação das relações na representação do modelo de origem em EMF no Eclipse, as classes *DependeeLink* e *DependerLink* possuem um atributo *linkName* que não possui nenhuma utilidade para as transformações. As relações bidireccionais estão representadas como duas relações

unidireccionais. Uma em cada direcção. Isto por imposição da ferramenta que não permite a representação deste tipo de relações.

Este metamodelo já contém a representação de herança ao nível dos actores. Trata-se de um melhoramento ao modelo inicial que permite a modelação de relações de IsA.

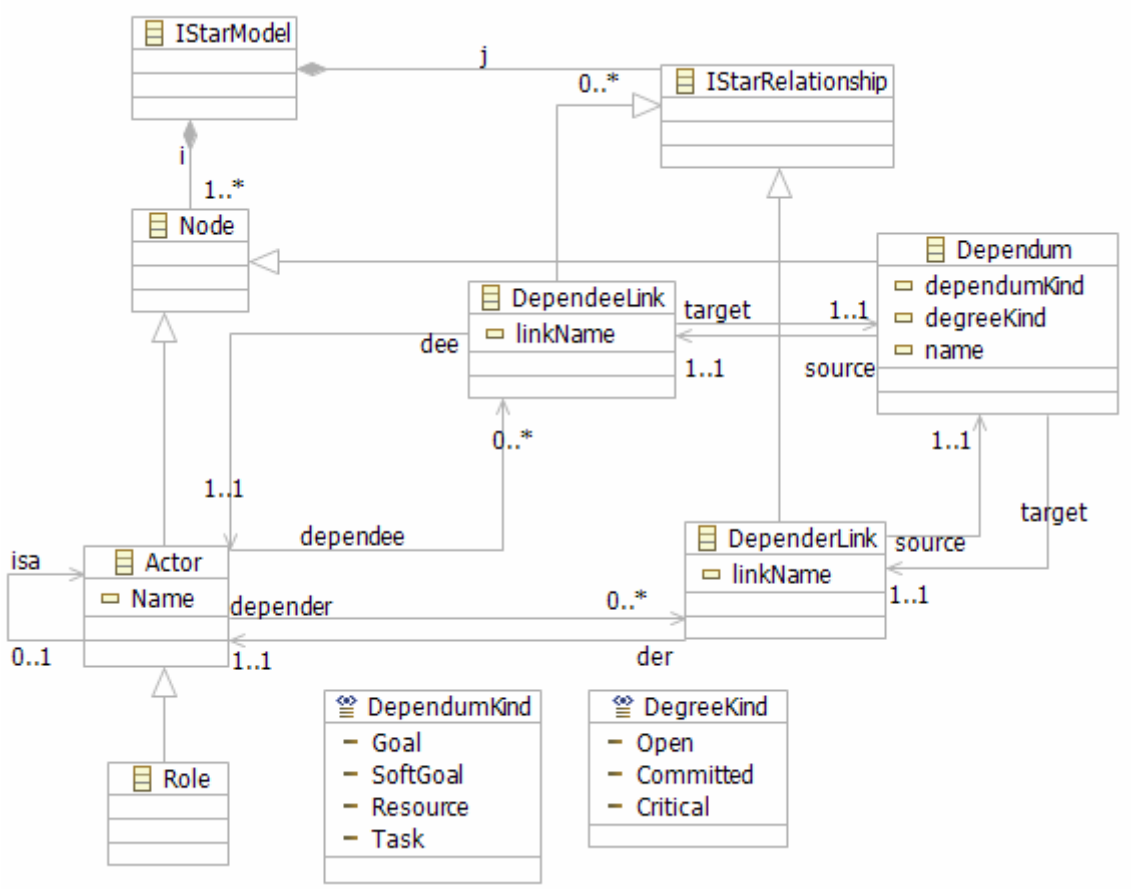

Figura 4.4 – Metamodelo do diagrama de dependência estratégica em Ecore.

Um dependum pode ainda ter vários graus de importância para o bom funcionamento do sistema. Pode ser *Open* quando a dependência não é relevante para o sucesso de uma operação. Será *Committed* se a sua falta puder levar a erros no resultado final de uma função ou operação mas, mesmo assim, o sistema não deixar de desempenhar, ainda que possivelmente mal, a sua função. Um dependum pode ainda ser denominado como *Critical* se a sua falta puser em causa o funcionamento do sistema no desempenho das suas funções. Esta informação não se encontrava modelada no diagrama arquitectural apresentado na figura 3.12 do capítulo anterior. Agora já é

possível visualizar esse tipo de informação no modelo de destino gerado pelas transformações ATL.

4.5 – Metamodelo Ecore do Diagrama Arquitectural do perfil de Agência

Um diagrama arquitectural de Agência é usado para modelar a visão estática dos diferentes módulos que constituem o sistema. O metamodelo deste diagrama serve para definir a estrutura existente entre os conceitos utilizados na descrição da arquitectura de um sistema. Tipicamente similar a um diagrama de classes, a figura abaixo representa a porção do metamodelo de agência descrito na secção 3.6.2, que reflecte as metaclasses e relações envolvidas na definição do diagrama arquitectural.

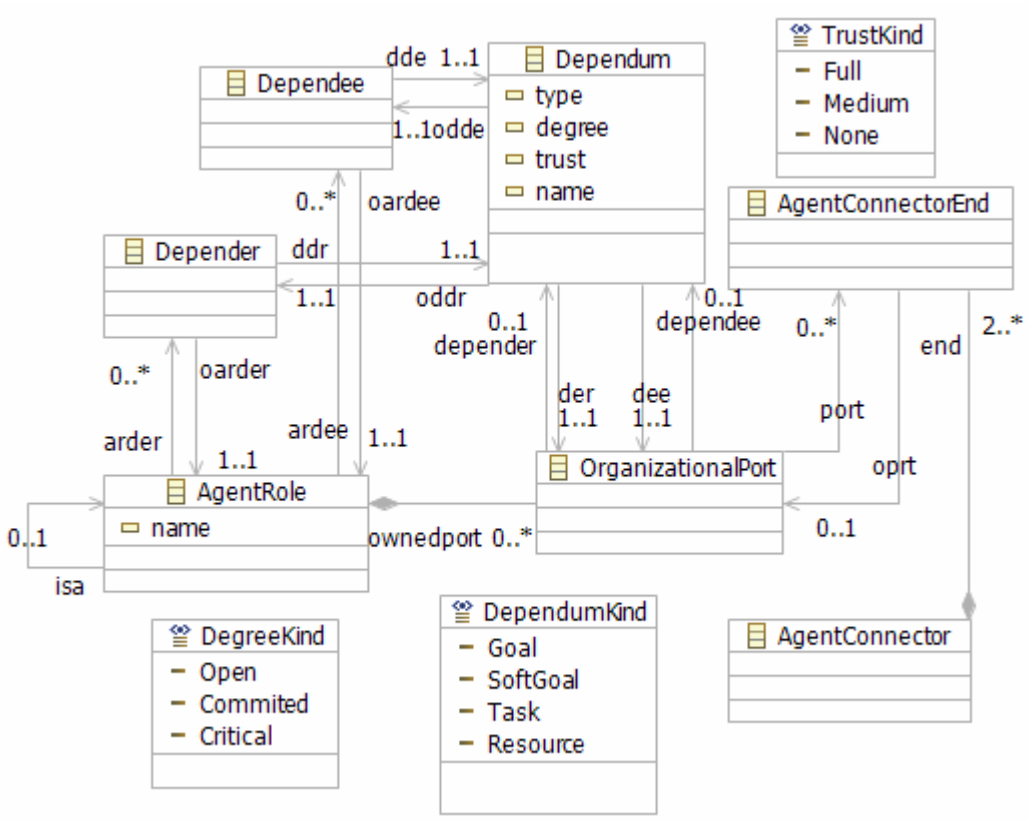

Figura 4.5 – Metamodelo do diagrama arquitectural em Ecore.

As relações aqui representadas, procuram reflectir a mesma realidade apresentada na figura 4.4. Este modelo é a versão Ecore do modelo MOF definido no Capítulo 3. As diferenças entre os dois modelos prendem-se com os mesmos aspectos referenciados aquando da apresentação da versão Ecore do metamodelo do diagrama SD.

4.6 – Regras de transformação

O processo de automatização dos mapeamentos indicados acima é feito recorrendo à ferramenta ATL apresentada no Capítulo 3.

Com os metamodelos definidos, é chegada a altura de apresentar as regras que concretizarão as transformações. Estas irão transformar um modelo SD do i*, num modelo arquitectural de Agência.

Como primeira regra (Listagem 4.1) temos o mapeamento de um actor para um *AgentRole*. Esta regra pega no conceito de Actor do i* e transforma-o directamente num AgentRole do Diagrama Arquitectural. Esta passagem é feita apenas dando o nome da entidade original à de destino. Esta transformação serve também para dar suporte às transformações que envolvam o conceito de herança. Desta forma, se um actor for uma especialização de outro terá essa informação na sua relação *isa* do metamodelo. Essa informação é passada automaticamente para o modelo de destino através já especificação isa <– ss.isa.

```
rule Actor2AgentRole {
      from
            ss : IStarSD!Actor
      to
            t : UMLArquitectural!AgentRole(
                  name <- ss.Name,
                  isa <- ss.isa
            )
}
```

Listagem 4.1 – Regra de transformação de Actor para AgentRole.

A segunda regra (Listagem 4.2) mapeia um *DependeeLink* do modelo SD para um *Dependee* no diagrama arquitectural. Tratando-se de duas classes sem atributos, a transformação consiste em mapear as ligações da classe de origem para a entidade de destino.

```
rule DependeeLink2Dependee {
      from
            s : IStarSD!DependeeLink
      to
            t : UMLArquitectural!Dependee(
                  dde <- s.target,
                  oardee <- s.dee
            )
      }
```

Listagem 4.2 – Regra de transformação de DependeeLink para Dependee.

Uma terceira regra (Listagem 4.3) transforma um *DependerLink* num *Depender*. Esta transformação ocorre na mesma linha de raciocínio da anterior.

```
rule DependerLink2Depender {
      from
            s : IStarSD!DependerLink
      to
            t : UMLArquitectural!Depender(
                  ddr <- s.source,
                  oarder <- s.der
            )
      }
```

Listagem 4.3 – Regra de transformação de DependerLink para Depender.

Como quarta transformação (Listagem 4.4), tem-se o mapeamento do dependum em i* para o dependum do diagrama arquitectural. O processo aqui é um pouco mais elaborado. A entidade Dependum contém vários atributos que importam tratar aquando da transformação. O atributo *type* toma o valor que o dependum original tiver em *dependumKind*. *Degree* terá o valor de *degreeKind* da entidade do metamodelo de origem. Tem-se um mapeamento directo no atributo *name* e um mapeamento das relações das classes nos respectivos metamodelos.

```
rule DependumSD2DependumArq {
      from
            s : IStarSD!Dependum
      to
            t : UMLArquitectural!Dependum(
              type <- s.dependumKind,
              degree <- s.degreeKind,
              name <- s.name,
              odde <- s.source,
              oddr <- s.target,
              trust <- if (s.degreeKind = 'Committed') then 'Medium'
                          else
                          if (s.degreeKind = '') then 'None'
                            else 'Full'
                          endif
                        endif
                  )
}
```

Listagem 4.4 – Regra de transformação de Dependum para Dependum.

Estas transformações cobrem as heurísticas que envolvem o diagrama de dependência estratégica e o diagrama arquitectural definidas em [Silva C., 2007]. Da

Tabela 4.1, são tratadas as heurísticas 1, 9, 10, 11 e 12. Apenas estas envolvem os conceitos presentes em ambos os diagramas alvo desta dissertação.

Para além destas, as heurísticas 4, 5, 14 e 16 da mesma tabela, também se reportam ao Diagrama Arquitectural. No entanto referem conceitos que estão ligado ao Diagrama de Razão Estratégica que não se encontra dentro do escopo desta dissertação. Por esse facto, as transformações aqui apresentadas não abrangem essas heurísticas.

Estas transformações vêm também concretizar as novas regras propostas na secção 4.3 do presente capítulo. Regras essas que visam completar as restantes, expostas na Tabela 1 que se referem ao diagrama arquitectural, mais concretamente ao diagrama SD, e que são automatizadas nas transformações aqui apresentadas.

4.7 – Sumário

Pelo que foi apresentado neste capítulo, podemos concluir que existem algumas limitações ao nível do mapeamento entre modelos arquitecturais descritos de acordo com o metamodelo i* e modelos arquitecturais descritos de acordo com o de Agência. Este trabalho descreve o processo que visa completar esse mapeamento.

Uma das lacunas encontradas no decorrer deste projecto foi ao nível das heurísticas que guiam essas mesmas transformações. Explorámos essas falhas e procurámos aperfeiçoar essas heurísticas. Estudaram-se as já existentes e propuseram-se mais três heurísticas capazes de servir as transformações de uma forma até agora não possível com as actuais.

Antes de concretizar essas transformações, tivemos que definir os metamodelos, de origem e destino, que servirão de base às mesmas. Esses metamodelos foram representados em Ecore por exigência da ferramenta. Estes metamodelos apresentam-se como partes do metamodelo geral. Na origem apresentámos a porção do metamodelo i* alusivo ao diagrama de dependência estratégica. No destino tínhamos a porção do metamodelo de Agência referente ao diagrama arquitectural. Actualmente, o formato Ecore apresenta ainda algumas limitações gráficas e isso notou-se nos diagramas gerados. Características como a bidireccionalidade ou a cardinalidade das relações exigiram alguma originalidade na elaboração desses diagramas.

Foram então introduzidas as regras de transformação. Com elas pretendemos automatizar a transformação de modelos SD do i* para modelos arquitecturais de Agência. Essas transformações abrangem todas as heurísticas, incluindo as novas, que englobam os conceitos presentes em ambos os modelos. Estas transformações foram

aplicadas ao exemplo usado para descrever a *framework* i*, o *Meeting Scheduler*. Aplicaram-se também ao exemplo do jornal electrónico, *E-News*, que é o modelo utilizado como caso de estudo no capítulo seguinte.

5 – Caso de estudo

Este capítulo descreve a nossa abordagem, concretizada no capítulo anterior. O propósito é demonstrar a viabilidade da nossa proposta. Para concretizar essa abordagem, utilizámos o exemplo E-news [Silva C., 2007].

5.1 – E-News

Neste exemplo pretende-se modelar o ambiente organizacional de um escritório de um jornal electrónico. Quando um utilizador pretende consultar uma notícia, pode aceder ao site do jornal mantido pelo Webmaster. Este é responsável pela actualização da informação lá presente. As versões publicadas do jornal são facultadas ao Webmaster pelo Editor Chefe. Este depende que cada Editor lhe forneça as notícias das diferentes áreas de interesse, que vão ser publicadas no jornal, preparadas de acordo com os princípios do periódico. Estes princípios, ou linhas de conduta, é acordado através de uma reunião entre os Editores e os Editor Chefe com o Chefe do jornal externo ao escritório. Para produzir uma edição é necessário que os princípios do jornal sejam dados a conhecer aos Editores, que são os responsáveis pela redacção de notícias de uma determinada categoria [Silva C., 2007].

Neste sistema, figura 5.1, existem vários Editores. Cada um responsável por uma área de interesse público. Por exemplo, podem existir Editores de desporto, economia, política ou artes. Cada Editor interage com um ou mais Repórteres Fotográficos na busca por notícias dentro de uma determinada categoria, sobre um assunto específico (por exemplo, futebol). As notícias elaboradas pelos Editores são então filtradas pelo Editor Chefe que, posteriormente, as envia para o Webmaster, para que este as publique no site.

5.2 – Desenho Arquitectural

Como o referido na secção 2.2, a fase de Desenho Arquitectural da metodologia Tropos serve o propósito de definir a arquitectura global do sistema em termos de subsistemas (actores), trocas de dados e fluxos de controlo (dependências).

Um estudo detalhado das fases de requisitos do Tropos já foi feito em [Silva C., 2007]. Desse estudo concluiu-se que as qualidades mais desejáveis para este sistema são a disponibilidade, a adaptabilidade e a segurança do sistema.

Para definir a arquitectura global do sistema, importa escolher o estilo organizacional que vai ser usado. Essa escolha é conduzida pelos *softgoals* identificados nas duas fases anteriores do Tropos. Esses *softgoals* são as qualidades preferenciais do nosso sistema. A escolha do estilo organizacional tem de ter isso em conta. Os estilos organizacionais foram estudados no Capítulo 2 e, segundo o Catálogo de Correlação usado pelo Tropos, o estilo *Joint-venture* (figura 2.5) foi o que apresentou mais vantagens perante o sistema em causa. Esse catálogo avalia todos os estilos organizacionais de acordo com critérios de qualidade de software devidamente identificados para arquitecturas que envolvem componentes autónomos e coordenados [Castro et al., 2002]. Esses critérios são, previsibilidade (1), segurança (2), adaptabilidade (3), coordenação (4), cooperação (5), disponibilidade (6), integridade (7), modularidade (8) e agregação (9). A tabela seguinte sumaria esta informação para os estilos organizacionais mais utilizados. A notação apresentada nessa tabela é a descrita em [Chung et al., 2000]. Segundo essa notação, +, ++, -, -- representam contribuições parcialmente positivas, suficientemente positivas, parcialmente negativas e suficientemente negativas respectivamente.

Tabela 5.1 - Catálogo de Correlação

	1	2	3	4	5	6	7	8	9
Flat Structure	--	--	-			+	+	++	-
Structure-in-5	+	+		+	-	+	++	++	++
Pyramid	++	++	+	++	-	+	--	-	
Joint-Venture	+	+	++	+	-	++		+	++
Bidding	--	--	++	-	++	-	--	++	
Takeover	++	++	-	++	-	+		+	+
Arm's-Length	-	--	+	-	++	--	++	+	
Hierarchical Contracting			+	+	+	+		+	+
Vertical Integration	+	+	-	+	-	+	--	--	--
Cooptation	-	-	++	++	+	--	-	--	

A figura 5.1 mostra uma arquitectura de um sistema multi-agente que reflecte a realidade do sistema alvo do nosso caso de estudo. Segundo o estilo organizacional *Joint-Venture*, existe um actor (Chief Editor) que coordena tarefas e gere os recursos partilhados entre os ouros actores parceiros (Editor, Webmaster e Repórter). Cada um dos actores parceiros controlam-se a si próprios numa dimensão local, e podem interagir directamente com outros actores para trocas de recursos. No entanto, o Chief Editor é o único responsável por operações estratégicas e de coordenação do sistema, e dos seus actores, a um nível global. Essa figura representa o modelo de origem na sua forma concreta.

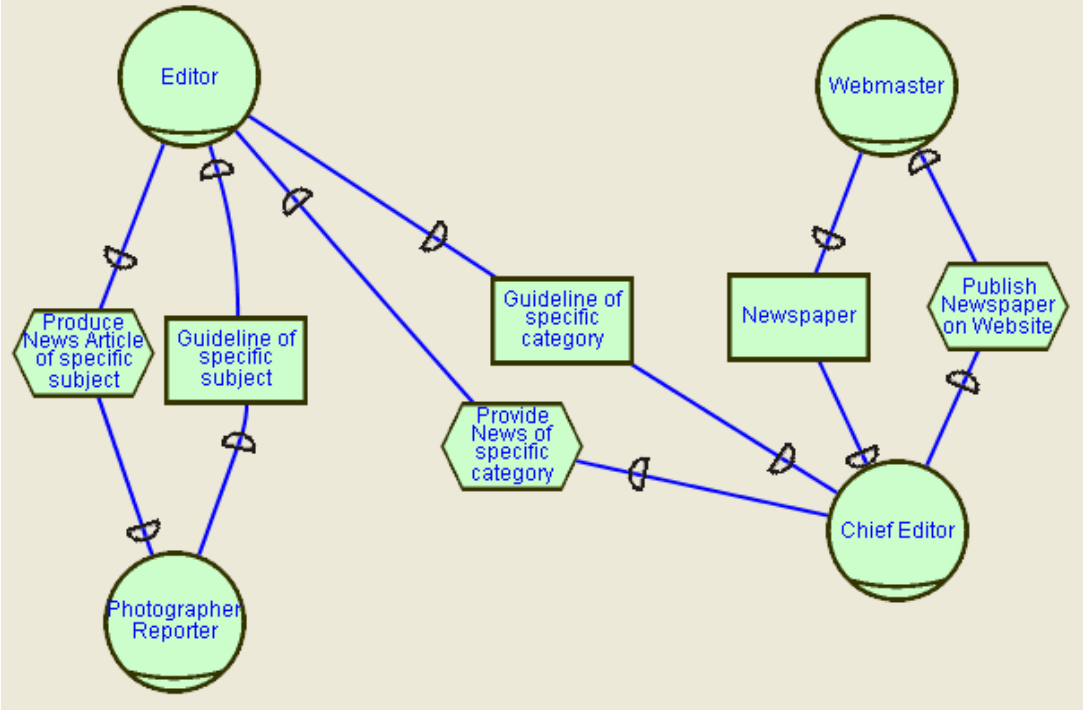

Figura 5.1 – Arquitectura do Sistema e-news – Estilo Joint Venture

5.3 - Mapeamentos

Uma vez identificados os componentes do sistema de marcação de reuniões, procura-se agora identificar os mapeamentos entre os elementos constituintes do diagrama SD da figura 3 e os componentes do Diagrama Arquitectural do metamodelo de Agência. Estes mapeamentos são feitos de acordo com as directrizes definidas no Capítulo 4.

Segundo as normas definidas até à data, os Agentes (ou Actores) Editor, Chief Editor, Webmaster e Photographer reporter são mapeados para AgentRoles no

Diagrama Arquitectural. Os recursos *Guideline of specific subject*, *Guideline of specific category* e *Newspaper* são enquadrados como um dependum do tipo *"Resource"* em UML. As tarefas *Produce News Article of specific subject*, *Provide News of specific category* e *Publish Newspaper on Website* são mapeadas para um dependum do tipo *"Task"*. Os recursos que os Agentes manipulam bem como as tarefas que estes desempenham, não se enquadram no panorama do diagrama arquitectural. Estas entidades não são assim mapeadas para outro tipo de classes, sendo apenas consideradas como um tipo particular de dependum.

5.3.1 - Automatização

Com o propósito de tornar mais clara como este modelo se representa, nomeadamente em formato UML (não é o modelo de destino), apresentamo-lo agora na sua forma abstracta como instâncias do metamodelo.

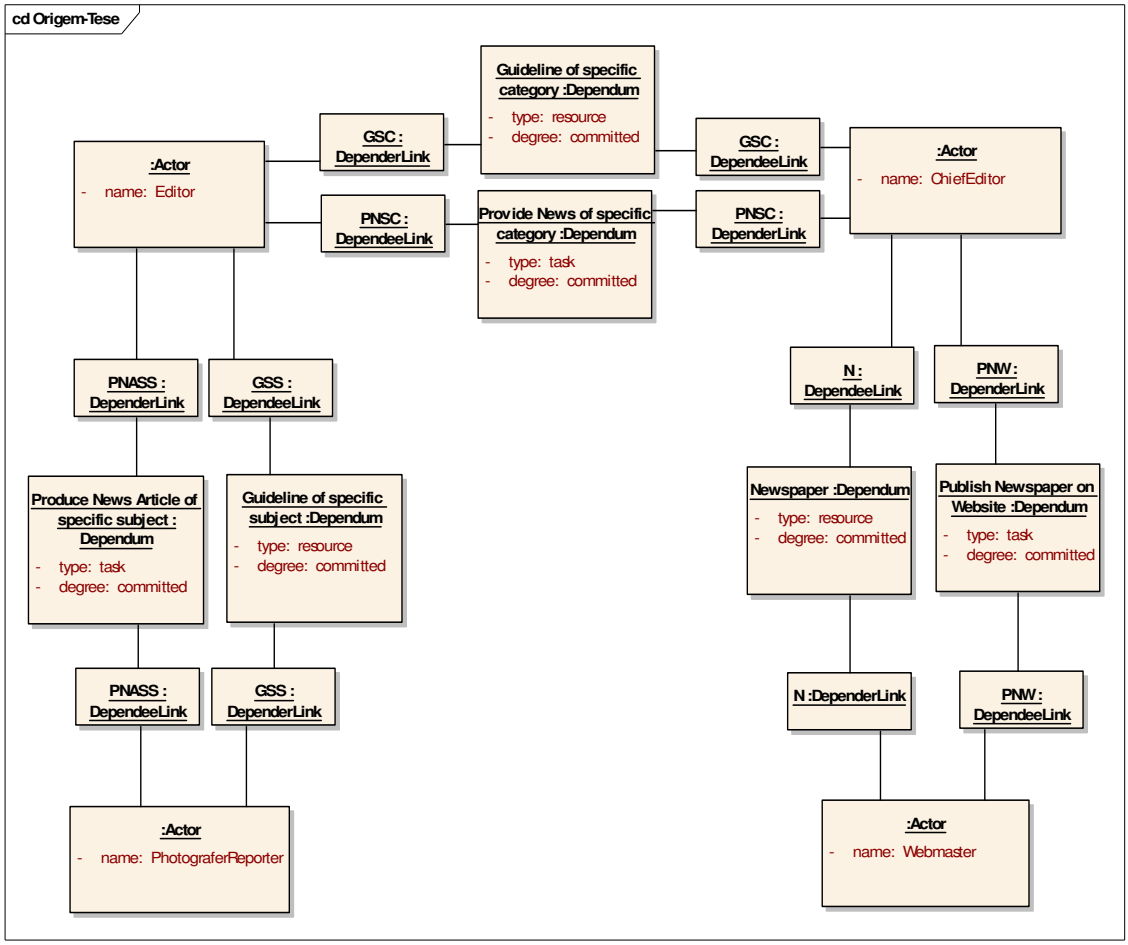

Figura 5.2 – Notação abstracta do modelo de origem – E-News.

Neste formato torna-se mais fácil identificar as entidades envolvidas em cada uma das relações entre objectos. Este modelo foi criado com base no da figura 5.1 e nos

seus mapeamentos. Assim, torna-se claro que o Actor Chief Editor se relaciona com o Editor e com o Webmaster. Nessas relações, o Chief Editor depende do Editor e do Webmaster para a realização das tarefas "*Provide News of Specific Category*" e "*Publish Newspaper on Website*" respectivamente. Para levarem a cabo estas tarefas, os actores Editor e Webmaster estão dependentes da disponibilidade dos recursos "*Guideline of Specific Category*" e "*Newspaper*".

Da mesma forma que os anteriores, os actores Editor e *Photographer Reporter* interagem entre si para a partilha de outros dois dependums: a tarefa "*Produce News Article of Specific Subject*" e o recurso "*Guideline of Specific Subject*".

A partir do metamodelo de origem, definido em Ecore no Capítulo 4, é possível criar um modelo de origem que represente o diagrama SD ilustrado na figura 5.1 e que esteja em conformidade com as restrições presentes no seu metamodelo. Esse modelo foi criado no ambiente EMF e é validado automaticamente pela ferramenta. A sua representação é a apresentada na figura 5.3.

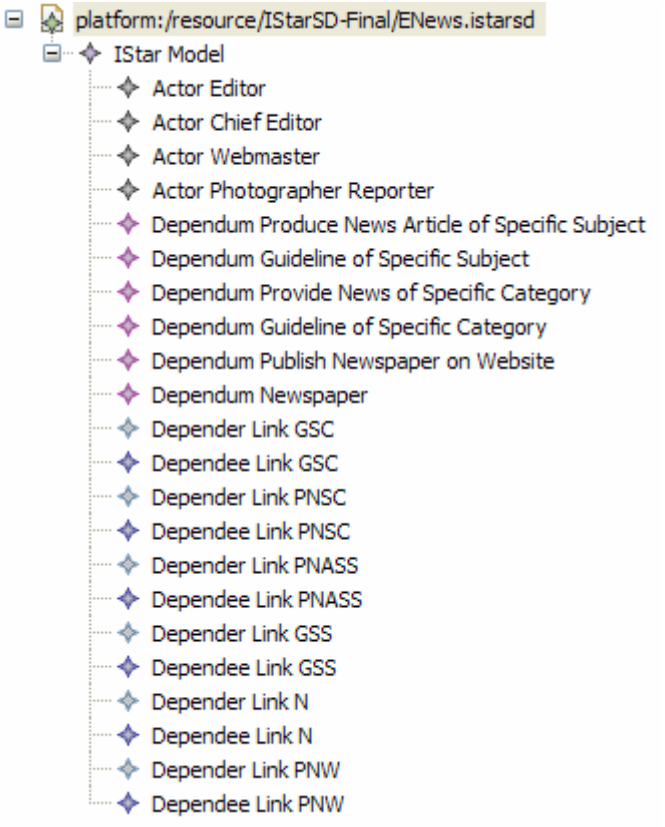

Figura 5.3 – Modelo de origem – E-News.

Este modelo não é, de facto, intuitivo mas possui as mesmas sintaxe e semântica que o modelo representado no Capítulo 4. Os elementos nele contidos foram dotados da

mesma estrutura sintáctica de forma a apresentarem a mesma semântica do modelo da figura 5.1.

Com o propósito de transformar o modelo da figura 5.3, representado na sua notação abstracta na figura 5.2, e na sua forma concreta na figura 5.1, num modelo que represente a mesma realidade, e que esteja em conformidade com o metamodelo do diagrama Arquitectural apresentado na figura 4.2, descreveram-se em ATL as transformações apresentadas no Capítulo 4. A aplicação dessas regras de transformação ao modelo inicial, deu origem ao modelo de destino apresentado nas Figuras 5.4, 5.5, 5.6, 5.7 e 5.8. Essas figuras representam o modelo de destino na sua forma concreta.

Nas figuras que se seguem importa tornar claro algumas abstracções presentes na geração de modelos usando ATL em ambiente EMF. Assim, os números que aparecem funcionam como apontadores para outros objectos do mesmo modelo. A numeração usada começa em zero. Posto isto, temos que o *AgentRole Editor* é o objecto zero, o *Chief Editor* é o objecto número um e por aí adiante.

Na figura 5.4 temos a representação EMF dos *AgentRoles* envolvidos no exemplo da Agenda de Reuniões. Temos quatro destas entidades. O *Editor*, o *Chief Editor*, o *Webmaster* e o *Photographer Reporter*. O primeiro *AgentRole* tem o papel de *depender* (da relação do metamodelo, *arder*) em duas relações sendo *dependee* noutras tantas (relação *ardee* do metamodelo). De igual modo, o *Chief Editor* desempenha o papel de *depender* em duas relações sendo *dependee* noutras duas. Por fim, os *AgentRoles Webmaster e Photographer Reporter,* exercem a posição de *depender* numa relação, sendo *dependee* noutra. A 5.4 abaixo mostra as entidades com os identificadores de zero a três.

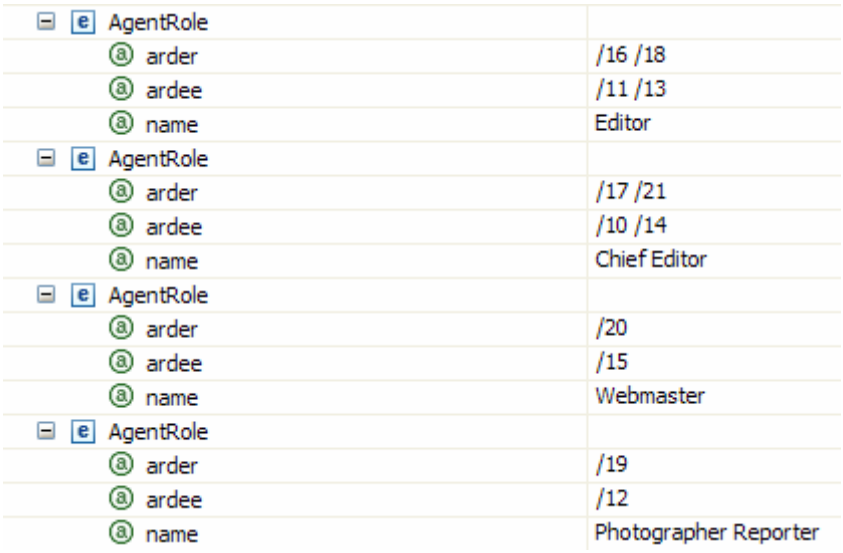

Figura 5.4 – Modelo de destino – Os AgentRoles (Entidades 0-3).

Neste formato, os dependums são representados de forma idêntica aos *AgentRoles*. Logo, num dependum podemos encontrar informação acerca do seu nome, e de quais as entidades que desempenham os papeis de *depender* e *dependee*. Por exemplo, o dependum "*Produce News Article of Specific Subject*", o depender é a entidade identificada pelo número 18 (terceira entidade na figura 5.8) que, por sua vez, aponta para o *AgentRole* identificado pelo número 0 (relação *oarder*), que é o primeiro elemento da figura anterior e representa o *Editor*. Quer isto dizer que o dependum "*Produce News Article of Specific Subject*" tem o *Editor* como *depender*. O *dependee* deste dependum é o representado na entidade com o número 12 que nos remete para um objecto *dependee* (terceira entidade da figura 5.7) que aponta (relação *oardee*) para um outro *AgentRole*, com identificador três. A entidade três é o *Photographer Reporter*. Assim, o *dependee* da relação é este último *AgentRole*. O mesmo raciocínio se aplica às restantes entidades presentes no modelo. A figura 5.5. representa as entidades de quatro a sete, e a figura 5.6 as entidades oito e nove. As figuras 5.7 e 5.8 apresentam as entidades de dez a quinze e de dezasseis a vinte e um respectivamente.

Dependum	
type	Task
degree	Committed
trust	Medium
oddr	/18
odde	/12
name	Produce News Article of Specific Subject
Dependum	
type	Resource
degree	Committed
trust	Medium
oddr	/19
odde	/13
name	Guideline of Specific Subject
Dependum	
type	Task
degree	Committed
trust	Medium
oddr	/17
odde	/11
name	Provide News of Specific Category
Dependum	
type	Resource
degree	Committed
trust	Medium
oddr	/16
odde	/10
name	Guideline of Specific Category

Figura 5.5 – Modelo de destino – Os Dependums (Entidades 4-7).

Dependum	
type	Task
degree	Committed
trust	Medium
oddr	/21
odde	/15
name	Publish Newspaper on Website
Dependum	
type	Resource
degree	Committed
trust	Medium
oddr	/20
odde	/14
name	Newspaper

Figura 5.6 – Modelo de destino – Os Dependums (Entidades 8 e 9).

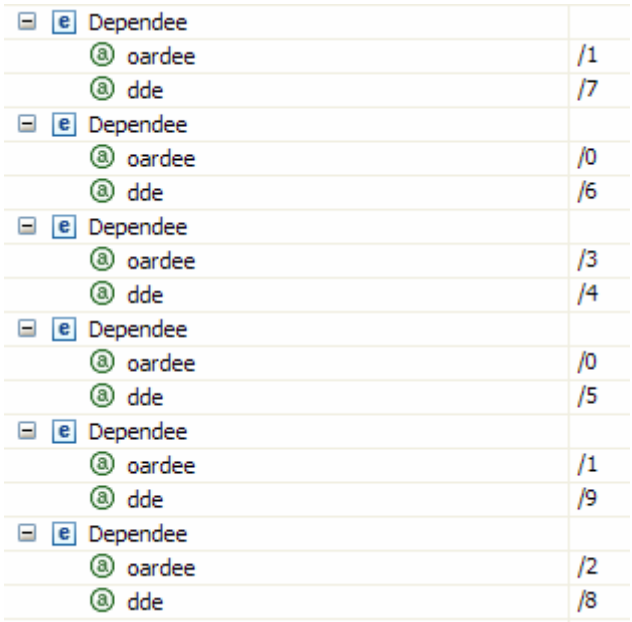

Figura 5.7 – Modelo de destino – Os Dependees (Entidades 10-15)

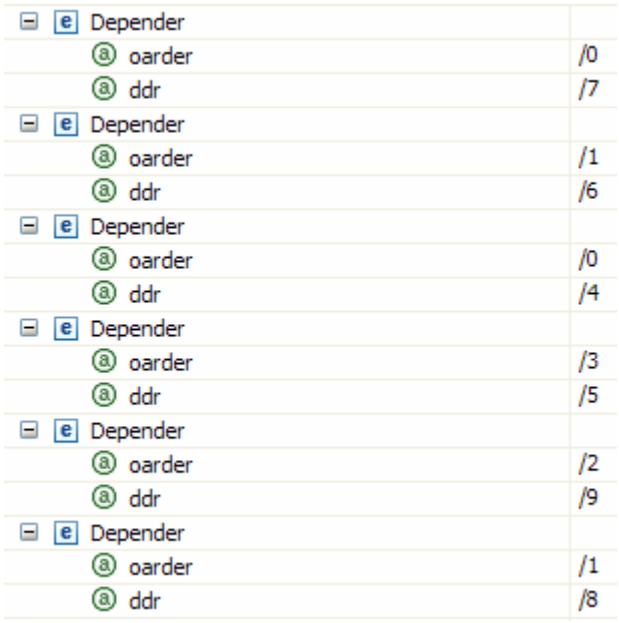

Figura 5.8 – Modelo de destino – Os Dependers (Entidades 16-21).

Reconhecendo que este formato não é intuitivo ao leitor, apresentamos a seguir uma parte do mesmo modelo representado na sua notação abstracta.

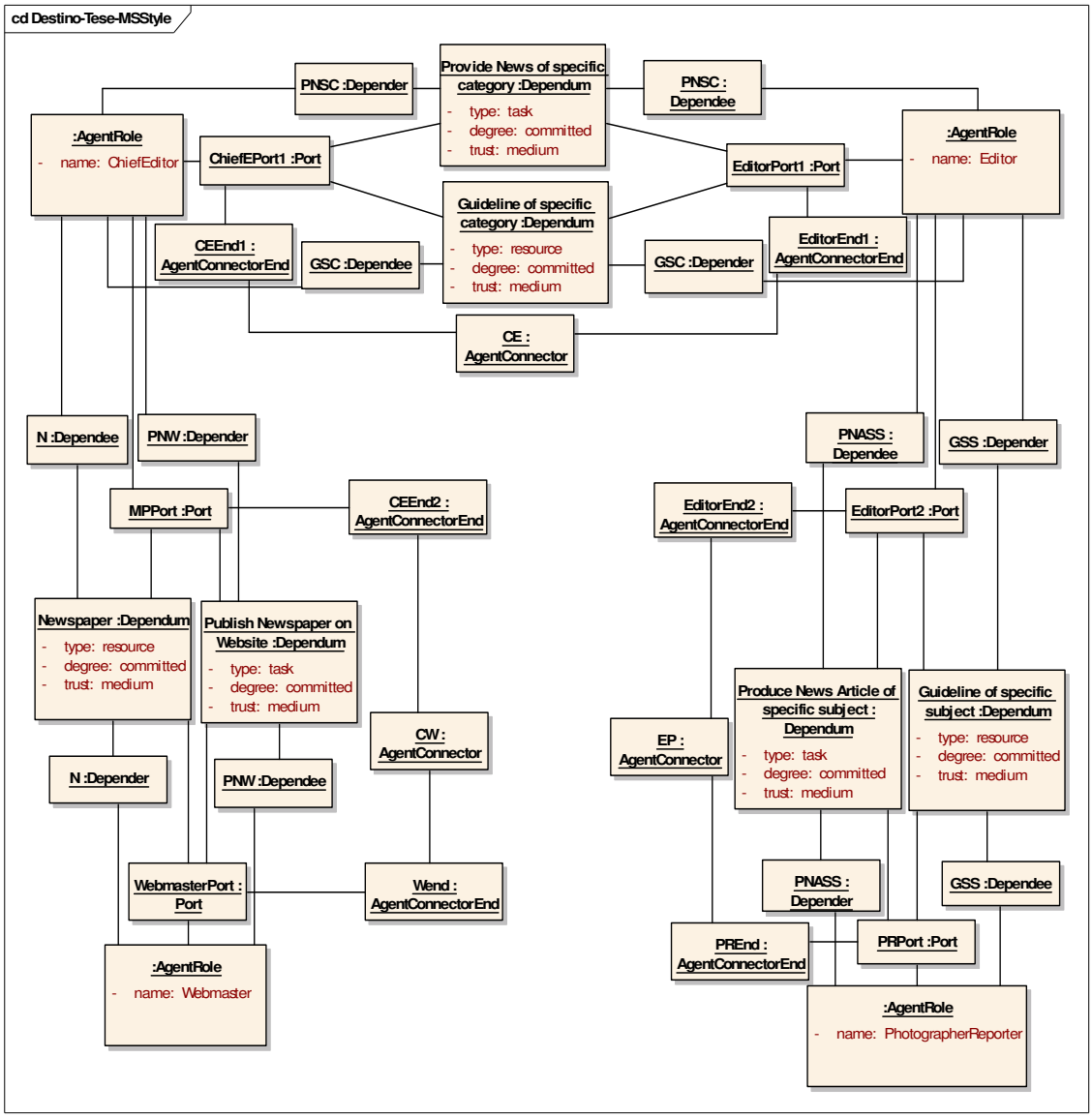

Figura 5.9 – Notação abstracta do modelo de destino – E-News.

A notação da figura 5.9 dá a perceber quais as entidades envolvidas em cada interacção definida no modelo. Também este modelo foi construído baseado nas relações existentes entre as entidades presentes no modelo gerado e ilustrado nas figuras 5.4, 5.5, 5.6, 5.7 e 5.8. Este diagrama da figura 5.9 modela exactamente a mesma realidade que a representada nas figura 5.1, 5.2 e 5.3, mas com algumas abstracções diferentes.

Desde logo se conseguem identificar algumas entidades que não existiam nos diagramas referentes ao modelo de origem. Conceitos como *Port*, *AgentConnector* e *AgentConnectorEnd* não têm correspondência com o diagrama de Dependência Estratégica. Estas abstracções visam detalhar um pouco mais a componente de

comunicação entre os Agentes. Este tipo de informação não é clara ao nível do diagrama SD.

5.4 – Trabalho relacionado

Em [Perini e Susi, 2006] encontrámos trabalhos que usam o Tropos como metodologia para construir sistemas multi-agente complexos. Estes trabalhos endereçam o papel das transformações de modelos à Engenharia de Software Orientada a Agentes (do inglês, AOSE – *Agent-Oriented Software Engineering*). Em particular, focam a transformação automática da decomposição de plano do Tropos para um diagrama de actividades do UML. Trata-se de uma abordagem baseada numa linguagem de transformação declarativa baseada nos conceitos de definição de padrões, regras de transformação e relações de rastreio (do inglês, *tracking relationships*). Essa linguagem não se apresenta tão intuitiva quanto a proposta nesta tese. Tal como nesta dissertação, também foi discutido o papel das transformações na metodologia Tropos, à luz do desenvolvimento orientado a modelos. Similarmente, em [Bertolini et al., 2006] sugere-se a ideia de um ambiente de desenvolvimento orientado a modelos que abarque todo o processo de desenvolvimento de software do Tropos.

O trabalho descrito em [Da Silva et al., 2006] foca o desenvolvimento de SMA usando uma abordagem orientada a modelos capaz de suportar a constante mudança de sistemas multi-agentes e que permita o mapeamento entre modelos de desenho e modelos de implementação. Tal como na nossa proposta, também aqui se descrevem os modelos de origem e de destino, os respectivos modelos e as transformações. Sendo o processo similar ao nosso, em [Da Silva et al., 2006] trabalha-se a um nível mais baixo de abstracção e apenas com conceitualizações de SMA.

Da mesma forma, mas agora na procura para detalhar as fases PSM e PIM do processo de desenvolvimento de SMA, o trabalho apresentado em [De Maria et al., 2004] define as transformações dos modelos que representam essas mesmas fases e, posteriormente conseguir a geração do código fonte.

Outro trabalho relevante nesta área é o descrito em [Pablo et al., 2006] onde se utiliza transformação de modelos entre modelos da Engenharia de Requisitos orientados a aspectos com o objectivo de automatizar o processo de derivação dos mesmos. Esta é uma abordagem mais focada nos aspectos e na forma como estes acompanham o processo de desenvolvimento de software.

Apesar de relevantes, quase todos estes trabalhos se focam nas fases mais avançadas do processo de desenvolvimento de software. O principal ponto de diferenciação entre a nossa abordagem e as demais enunciadas nesta secção está relacionado com isso mesmo. A nossa abordagem trata das fases iniciais – de requisitos e desenho arquitectural. Para além disso na nossa proposta procuramos conciliar duas *frameworks* diferentes na forma dos seus metamodelos: o metamodelo i* e o metamodelo de Agência, procurando harmonizar o melhor que cada uma tem para oferecer.

5.5 – Sumário

Este capítulo focou uma aplicação prática da nossa proposta. Apresentou-se a especificação de um sistema de gestão de conteúdos de um jornal electrónico em i* na sua notação concreta e abstracta, bem como no formato Ecore utilizado pela ferramenta ATL utilizada para levar a cabo as transformações. O modelo na sua notação concreta foi elaborado com base no trabalho feito em [Silva C., 2007]. Pegámos nas conclusões retiradas após a realização das duas primeiras fase do Tropos (*Early* e *Late Requirements*) e, a partir desse ponto, detalhámos a fase de desenho arquitectural e trabalhámos sobre o modelo daí resultante. Traduzimos esse modelo para o formato da ferramenta e aplicámos as regras de transformação, que concretizam as heurísticas já existentes e o novo conjunto de heurísticas, ao modelo SD do exemplo em causa. Obteve-se como resultado um modelo final que representa a mesma realidade mas de acordo com as abstracções presentes no diagrama arquitectural de Agência.

Ao modelo obtido por aplicação das transformações, traduzimo-lo para um formato UML, de forma a potenciar a sua compreensão. Serviu essa tradução fiel das relações presentes no modelo gerado para comprovar que realmente a semântica dos dois modelos, o de origem e o de destino, são iguais. A diferença reside nas abstracções de cada um dos modelos. Um diagrama arquitectural possui algumas construções que não existem no diagrama SD mas que no fundo, com essas novas abstracções, modelam a mesma realidade.

Apresentamos agora as nossas conclusões e os trabalhos futuros que podem ser desencadeados após a realização desta dissertação.

6 – Conclusões

O estudo feito para a elaboração desta dissertação permitiu aprender muito acerca de SMA e MDD, matérias sobre as quais já possuía algumas (poucas) bases, que foram reforçadas durante este período de investigação. Mas a principal contribuição para a minha formação académica está relacionada aos conceitos propostos pela *framework* i*, que até agora me era completamente desconhecida (este foi o primeiro contacto com i* através do estudo do projecto Tropos), e ainda relativamente à transformação de modelos. Durante o meu percurso académico nunca tinha trabalhado nesta área, que considero interessante, na medida em que ajuda a simplificar a implementação e manutenção de software, como discutido nesta dissertação. O pouco conhecimento que tinha da área foi o maior obstáculo encontrado mas também um foco de interesse e um corte na monotonia.

Ao longo deste documento abordámos muitos conceitos importantes como engenharia de requisitos, metodologias orientadas a objectivos, agentes, desenvolvimento orientado a modelos, metamodelação, UML e arquitectura de software. Todos estes conceitos estarão envolvidos na realização do trabalho proposto nesta dissertação, cujo maior objectivo é beneficiar a engenharia de sistemas multi-agentes. Portanto, este relatório fornece uma base de consulta essencial dos principais conceitos envolvidos na realização do trabalho proposto e relata a forma como eles cooperam para alcançar o objectivo final apresentado nesta dissertação.

6.1 - Contribuições

O trabalho desenvolvido durante a realização desta dissertação apresenta um conjunto de contribuições.

1. Representa um passo rumo à automatização do processo de transformação de modelos arquitecturais descritos em i* para modelos arquitecturais descritos num perfil de Agência UML. Esse processo foi discutido ao longo da dissertação e envolve a utilização de MDD.

2. O processo de desenvolvimento de sistemas multi-agente, apresentado neste relatório por intermédio do projecto Tropos, formou a base de trabalho desta dissertação. A partir dela desenvolvemos um processo de implementação SMA usando MDD. Apresentámos uma abordagem assente na linguagem de transformação ATL em ambiente Eclipse (EMF), baseada em transformações entre os metamodelos de origem e destino dos diagramas acima enunciados. Implementámos ainda um conjunto de regras de transformação, utilizando a linguagem de transformação ATL.
3. O uso de MDD no nosso método dota a abordagem proposta de uma série de vantagens inerentes às metodologias orientadas a modelos. Primeiro porque se parte de um modelo totalmente independente de plataforma, como é o diagrama de dependência estratégica do i*). Segundo, porque tratando-se de modelos de modelos independentes de plataforma (os ditos PIM), estes são portáveis e podem ser reutilizados na geração de diferentes modelos computacionais aplicados sobre diferentes plataformas de implementação. Assim, a nossa abordagem de automatização da fase de desenho arquitectural do projecto Tropos dota esta fase das vantagens decorrentes do uso de MDA. Entre elas, destaque para as quatro principais já referidas, logo no Capítulo 1: portabilidade, produtividade, interoperabilidade, manutenção e documentação.

   Foi utilizado um caso de estudo para demonstrar a razoabilidade desta proposta. Nele foi definido o modelo de origem em conformidade com o seu metamodelo e foram aplicadas as regras de transformação definidas para o efeito. Antes foram definidos os metamodelos dos modelos de entrada e de saída que serviram de base às transformações. Essas transformações foram implementadas usando a ferramenta ATL, que produz um modelo final descrito em XMI (do inglês, *XML Metadata Interchange*). O uso desta ferramenta de geração automática de modelos diminui a complexidade inerente ao uso de XML (do inglês, *eXtensible Markup Language*).
4. Com esta dissertação saiu reforçado o conjunto de heurísticas que auxiliam os engenheiros de software na construção de modelos arquitecturais a partir do modelo de dependência estratégica da *framework* i*. Os modelos arquitecturais gerados podem ter um grau de complexidade elevado dependendo do domínio da aplicação.

5. A automatização proposta pretende tornar mais expedito o mapeamento entre os conceitos do metamodelo i* e os do metamodelo de Agência, de forma a facilitar a definição da fase de desenho arquitectural do ciclo de vida do processo de desenvolvimento de software.

Esta investigação foca na automatização do processo de mapear os conceitos representados em i* para as abstracções presentes no metamodelo de Agência. A principal característica do Tropos é que ambiciona abranger todo o ciclo de vida do processo de desenvolvimento de software. É, de facto, uma metodologia bastante completa nesse sentido e, até à data apresenta como principal dificuldade o facto de usar como base uma *framework* organizacional (a do i*), que não é adequada às fases finais deste processo.

Com esta abordagem podemos começar materializar a ideia de que é possível juntar o melhor de dois mundos. Podemos beneficiar da capacidade da framework i* para modelar estilos organizacionais, e assim termos uma metodologia bastante apta para as fases de *early* e *late requirements*. Podemos também usufruir das capacidades de modelação de arquitectura que o UML possui para continuar o processo a partir da fase de desenho arquitectural.

6.2 – Limitações

No âmbito do objectivo desta dissertação podemos apontar duas limitações ao trabalho realizado.

1. A primeira está relacionada com facto de a representação dos modelos de origem e de destino, assim como com a dos metamodelos alvo deste projecto, não possuírem um formato intuitivo ao leitor. Nessa perspectiva, será interessante uma abordagem a este problema utilizando um ambiente, ou ferramenta, capaz de dotar os modelos de propriedades gráficas mais amigáveis.
2. A segunda limitação é que, para os modelos ficarem num formato perceptível para o leitor, precisamos desenhar um modelo de notação abstracta (instâncias do metamodelo) de uma forma informal.

Qualquer uma destas limitações servem de motivação para o trabalho futuro a realizar nesta área, e que é apresentado de seguida.

6.3 – Trabalho Futuro

Como foi visto ao longo deste documento, os metamodelos alvo desta dissertação são constituídos por outros diagramas que não apenas o de dependência estratégica e o diagrama arquitectural. Assim, em i*, para além do SD existe também o diagrama de razão estratégica. Já o metamodelo de agência divide-se em vários diagramas: o de Comunicação, o Ambiental, o Intencional, o Racional e o de Plano. No futuro, qualquer um destes devem ser merecedores de uma análise de mapeamento, com o mesmo grau de detalhe que esta. Esse estudo pode ser feito não só relativamente ao diagrama de dependência estratégica como também relativamente ao diagrama de razão estratégica do metamodelo i*. O processo de automatização das transformações entre os metamodelos de i* e de Agência, não se esgota no mapeamento entre os diagramas de dependência estratégica do primeiro, e arquitectural do segundo. Seria interessante mapear os conceitos presentes em todos os diagramas, em ambas as *frameworks*. Isto reforça a ideia de que estamos perante um trabalho que exige continuidade no que respeita ao processo de mapear os conceitos i* para UML.

Para além da definição de heurísticas e realização de transformações para os diversos diagramas enumerados acima, importa também potenciar o uso de uma ferramenta, ambiente, capaz de produzir diagramas mais intuitivos graficamente. Um ambiente de trabalho bastante próximo do utilizado poderá ser o GMF.

7 – Bibliografia

[Alencar et al., 2008] Alencar, F., Silva, C., Lucena, M., Castro, J. F. B., Monteiro, C., Ramos, R., Santos, E. “Improving the understandability of i* models”. In: 10th International Conference on Enterprise Information Systems (ICEIS 08), 2008, Barcelona. June 12 - 16, 2008.

[Baresi e Heckel, 2002] Luciano Baresi and Reiko Heckel, Tutorial Introduction to Graph Transformation: A Software Engineering Perspective, *Proceedings of the First International Conference on Graph Transformation*, Barcelona, 2002.

[Bertolini et al., 2005] Davide Bertolini, Anna Perini, Angelo Susi and Haralambos Mouratidis, The Tropos visual modeling language. A MOF 1.4 compliant meta-model, 2005.

[Bertolini et al., 2006] A Tropos Model-Driven Development Environment. Davide Bertolini, Loris Delpero, John Mylopoulos, Aliaksei Novikau, Alessandro Orler, Loris Penserini, Anna Perini, Angelo Susi, and Barbara Tomasi. Trento, Italy, 2006.

[Bézivin et al., 2003] Jean Bézivin, Erwan Breton, Grégoire Dupé, Patrick Valduriez, The ATL Transformation-based Model Management Framework. IRIN, Université de Nantes, 2003.

[Bézivin et al., 2006] J. Bézivin, R. Chevrel, H. Brunelière, A. Jossic, W. Piers, F. Jouault, ModelExtractor an Automatic Parametric Model Extractor, 2006.

[Brown, 2006] A. W. Brown, S. Iyengar, S. Johnston, A Rational approach to model driven development, 2006.

[Bubenko, 1993] J. A. Bubenko, Extending the Scope of Information Modeling, Proc. 4th Int. Workshop on the Deductive Approach to Information Systems and Databases, Lloret-Costa Brava, Catalonia, Sept. 20-22, 1993, p. 73-98.

[Bubenko, 1995] J. A. Bubenko, Challenges in Requirements Engineering, *Proc. 2nd IEEE Int. Symposium on Requirements Engineering*, York, England, March 1995, p. 160-162.

[Castro et al., 2002] Jaelson Castro, Manuel Kolp, John Mylopoulos, Towards Requirements-Driven Information Systems Engineering: The Tropos Project, 2002.

[Chitchyan et al., 2005] Ruzanna Chitchyan, Awais Rashid, Pete Sawyer, Alessandro, Garcia (ULANC), Mónica Pinto Alarcon (UMA), Jethro Bakker, Bedir Tekinerdogan (UT), Siobhán Clarke, Andrew Jackson (TCD), "AOSD-Europe-ULANC-9", Milestone No: D11, Version 1.0, Survey, 2005.

[Chung e Yu, 1998] Chung, L. and Yu, E., Achieving System-Wide Architecture Qualities, OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, California, January 1998.

[Chung et al., 1995] Chung, L., B. Nixon and Yu, E., Using Non-Functional Requirements to Systematically Select, Among Alternatives in Architectural Design, Proc., 1st International Workshop on Architectures for Software Systems, Seattle, April 1995 24-28, p. 31-43.

[Chung et al., 2000] L. Chung, B. Nixon, E. Yu, J. Mylopoulos, Non- Functional Requirements in Software Engineering, Kluwer Publishing, Dordrecht, 2000.

[Chung, 1998] Chung, L., Architecting Quality Using Quality Requirements, *Proc. KUST*, Oct. 22-24, Vienna, Virginia, 1998.

[Curtis et al., 1988] B. Curtis, H. Krasner and N. Iscoe, A Field Study of the Software Design Process for Large Systems, *Communications of the ACM*, 31(11), 1988, p. 1268-1287.

[Da Silva et al., 2006] A MDE-Based Approach for Developing Multi-Agent Systems. Viviane Torres da Silva, Beatriz de Maria, Carlos J. P. de Lucena Proceedings of the Third Workshop on Software Evolution through Transformations: Embracing the Change (SeTra 2006).

[Dardenne et al., 1993] A. Dardenne, A. van Lamsweerde and S. Fickas, Goal- Directed Requirements Acquisition, Science of Computer Programming, 20, 1993, p. 3-50.

[De Maria et al., 2004] Usando MDA no Desenvolvimento de Sistemas Multi-Agentes. Beatriz Alves De Maria, Viviane Torres da Silva, Carlos José Pereira de Lucena. Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 2004.

[Finkelstein e Sommerville, 1996] A. Finkelstein and I. Sommerville, The Viewpoints FAQ, *BCS/IEE Software Engineering Journal*, vol. 11, 1996.

[Gazendam e Jorna, 1993] Gazendam, H.W.M. and Jorna, R.J., Theories about architecture and performance of multi-agent systems, *III European Congress of Psychology*, Tampere, Finland, July 1993.

[Gotel e Finkelstein, 1994] O.C.Z. Gotel and A.C.W. Finkelstein, An Analysis of the Requirements Traceability Problem, *Proc. IEEE Int. Conf. on Requirements Engineering*, Colorado Springs, April 1994, p. 94-101.

[Gross e Yu, 2001] Gross, D. and Yu, E., From Non-Functional Requirements to Design through Patterns, Springer-Verlag London Limited, Requirements Eng 6:18–36, 2001.

[Grudin, 1988] J. Grudin, Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces, *Proc. Conference on Computer-Supported Cooperative Work,* 1988, p. 85-93.

[Hailpern e Tarr, 2006] B. Hailpern, P. Tarr, Model-driven development: The good, the bad, and the ugly. IBM SYSTEMS JOURNAL, VOL 45, NO 3, 2006

[Hammer e Champy, 1993] M. Hammer and J. Champy, Reengineering the Corporation: A Manifesto for Business Revolution, HarperBusiness, 1993.

[Hatch e Cunliffe. 2006] Mary Jo Hatch and Ann L. Cunliffe. Organization Theory - Modern, Symbolic, and Postmodern Perspectives Second Edition, 2006.

[Jacobson, 2003] I. Jacobson, Use Cases - Yesterday, Today, and Tomorrow, *The Rational Edge*, 2003.

[Jouault e Kurtev, 2006] Frédéric Jouault, Ivan Kurtev, Transforming Models with ATL. ATLAS Group (INRIA & LINA, University of Nantes), p. 128-138. Springer, Berlin, 2006.

[Kavakli, 2004] Kavakli, E., Modeling organizational goals: Analysis of current methods, *Proceedings of the 2004 ACM Symposium on Applied Computing, Nicosia*, CY, March 2004, ISBN:1-58113-812-1, p. 1339 – 1343.

[Kleppe et al., 2007] A. Kleppe, J. Warmer, Wim Bast, MDA explained The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2007.

[Koch, 2006] Nora Koch, Transformation Techniques in the Model-Driven Development Process of UWE. Workshop proceedings of the sixth international conference on Web engineering, Califórnia, 2006.

[Kolp et al., 2002] Kolp, M., Giorgini, P., Mylopoulos, J.: Information Systems Development through Social Structures. 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), Ishia, Italy, July (2002)

[Kotonya e Sommerville, 1997] Kotonya, G., Sommerville, I.: Requirements Engineering – Processes and Techniques. Chichester, John Willy & Sons, 1997

[Lamsweerde et al., 1995] A. Van Lamsweerde, R. Darimont and Ph. Massonet, Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons

Learnt, *Proceedings of 2nd IEEE Int. Symposium on Requirements Engineering*, York, England, March 1995, p. 194-203.

[Lamsweerde, 2000] A. van Lamsweerde, Requirements condemned engineering in the year 00: A research perspective, International Conference on Software Engineering. Irlanda, 2000.

[Lamsweerde, 2001] Lamsweerde, A. van., Goal-Oriented Requirements Engineering: A Guided Tour, *Proceedings RE'01, 5th International Symposium on Requirements Engineering*. Toronto, Canada. August 2001.

[Lara e Guerra, 2005] Juan de Lara, Esther Guerra, Formal Support for Model Driven Development with Graph Transformation Techniques, 2005.

[Lengyel, 2006] László Lengyel, ONLINE VALIDATION OF VISUAL MODEL TRANSFORMATIONS, Budapest University of Technology and Economics Department of Automation and Applied Informatics, Ph.D. Thesis, Budapest, 2006.

[Loucopoulos e Karakostas, 1995] Loucopoulos, P. and Karakostas, V., System Requirements Engineering, McGraw-Hill Book Company, 1995.

[Madari, 2007] Madari, I., Lengyel, L., Levendovszky, T., Modeling the User Interface of Mobile Devices with DSLs. 8th International Symposium of Hungarian Researchers on Computational Intelligence and Informatics, Budapest, Hungary, 2007.

[Mens et al., 2004] Mens, T., Czarnecki, K., Van Gorp, P., A Taxonomy of Model Transformations, *Proc. Dagstuhl Seminar 04101*, 2004.

[Merks, 2004] The Eclipse Modeling Framework - Introducing Modeling to the Java™ Technology Mainstream. Ed Merks, Ph.D. IBM Canada, 2004 www.eclipse.org/emf

[Mészáros et al., 2007] Tamás Mészáros, Gergely Mezei, Hassan Charaf, A General Purpose Model Visualization Environment, 8th International Symposium of Hungarian Researchers on Computational Intelligence and Informatics, 2007.

[Mezei et al., 2006] Gergely Mezei, László Lengyel, Tihamér Levendovszky, Hassan Charaf, A Model Transformation for Automated Concrete Syntax Definitions of Metamodeled Visual Languages. Proceedings of the Second International Workshop on Graph and Model Transformation (GraMoT 2006).

[Minsky e Muarata, 2004] Minsky, N. and Muarata, T. "On Manageability and Robustness of Open Multi-Agent Systems", In: Lucena, C. et al. (eds.): Soft. Eng. for Multi-Agent Systems II: Research Issues and Practical App.. LNCS, Vol. 2940, Springer-Verlag, p. 189 – 206, 2004.

[OMG, 2001] OMG Architecture Board ORMSC. Model driven architecture (MDA). OMG document number ormsc/2001-07-01, available from www.omg.org, July 2001.

[OMG, 2005] Object Management Group (OMG), "MOF QVT Final Adopted Specification - http://www.omg.org/docs/ptc/05-11-01.pdf," 2005.

[Pablo et al., 2006] Towards MDD Transformations from AO Requirements into AO Architecture. Pablo Sánchez, José Magno, Lidia Fuentes, Ana Moreira, and João Araújo. LNCS 4344, Springer-Verlag Berlin Heidelberg, p. 159–174, 2006.

[Perini e Susi, 2006] A. Perini and A. Susi, Automating Model Transformations in Agent-Oriented Modelling, Trento-Povo, Italy p. 167-178, 2006

[Ponsard, 2004] C. Ponsard, P. Massonet, A. Rifaut and J.F. Molderez, A. van Lamsweerde and H. Tran Van, Early Verification and Validation of Mission Critical Systems, 2004.

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., Booch, G., The Unified Modeling Language – Reference Manual. Addison Wesley, 1999.

[Santos, 2007] J. Santos: Requisitos Aspectuais em Linhas de Produto usando MDD Relatório do Projecto Final de Curso, 2007.

[Shaw e Garlan, 1996] Shaw, M., and Garlan, D., Software Architecture: Perspectives on an Emerging Discipline, New Jersey, 1996.

[Silva C. et al., 2006a] Carla Silva, João Araújo, Ana Moreira, Jaelson Castro, Patrícia Tedesco, Fernanda Alencar and Ricardo Ramos, Modeling Multi-Agent Systems using UML, 2006.

[Silva C. et al., 2006b] Carla Silva, João Araújo, Ana Moreira, Jaelson Castro, Fernanda Alencar and Ricardo Ramos, ORGANIZATIONAL ARCHITECTURAL STYLES SPECIFICATION, 2006.

[Silva C., 2003] Silva C., Detalhando o projeto arquitetural no desenvolvimento de software orientado a Agentes: O caso TROPOS. Centro de Informática da Universidade Federal de Pernambuco, Brasil, 2003.

[Silva C., 2007] Silva C., Separating Crosscutting Concerns in Agent Oriented Detailed Design: The Social Patterns Case. Centro de Informática da Universidade Federal de Pernambuco, Brasil, 2007.

[Silva e Leite, 2005] Lyrene Fernandes da Silva, Júlio César Sampaio do Prado Leite, Uma Linguagem de Modelagem de Requisitos Orientada a Aspectos, 2005.

[Susi et al., 2005] A. Susi, A. Perini, P. Giorgini, and J. Mylopoulos. The Tropos Metamodel and its Use. *Informatica*, 29(4):401–408, Italy, 2005.

[Vajk et tal., 2007] Tamás Vajk, Gergely Mezei, Hassan Charaf, Architecture of an In-Memory Transformation Engine, 8th International Symposium of Hungarian Researchers on Computational Intelligence and Informatics, 2007.

[Varró e Pataricza, 2004] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor (eds.), Proc. UML 2004: 7th International Conference on the Unified Modeling Language, vol. 3273 of LNCS, p. 290–304. Springer, Lisbon, Portugal, 2004.

[VMTS, 2008] Visual Modeling and Transformation System: http://vmts.aut.bme.hu

[Wautelet et al., 2005] Wautelet, Y., Kolp, M. and Achbany, Y.: S-Tropos – An Iterative SPEM-Centric Software Project Management Process. Technical Report, Available at http://www.isys.ucl.ac.be/staff/youssef/Articles/WP_SPEM.pdf, 2005.

[Wooldridge, 2002] Wooldridge, M., An Introduction to Multiagent Systems. England. p. 15-103, ISBN: 0-471-49691-X, 2002.

[Yu, 1993] E. Yu, Modelling Organizations for Information Systems Requirements Engineering, *Proceedings of First IEEE Symposium on Requirements Engineering*, San Diego, Calif., January 1993, p. 34-41.

[Yu, 1995] E. Yu, *Modeling Strategic Relationships for Process Reengineering*, Ph.D. thesis, also Tech. Report DKBS-TR- 94-6, Dept. of Computer Science, University of Toronto, 1995.

[Yu, 1997] E. Yu, Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, presented at Requirements Engineering, Washington D.C., USA., 1997.

[Zambonelli et al., 2003] Zambonelli, F., Jennings, N., Wooldridge, M. "Developing Multiagent Systems: the Gaia Methodology", In ACM Transactions on Software Engineering and Methodology, 12, 3, p. 317 – 370, 2003.